Physical aggregated objects and dependability
Objets physiques agrégés et sûreté de fonctionnement

Fabrice Ben Hamouda

Lundi 13 Septembre 2010

Résumé

Ce rapport présente mon travail durant mon stage de 3 mois dans l'équipe ACES (Ambient Computing and Embedding Systems) au centre de recherche INRIA Rennes — Bretagne Atlantique. Ce stage a été supervisé par Michel Banâtre, directeur de l'équipe ACES et par Fabien Allard, ingénieur de SenseYou, jeune entreprise innovante issue des recherches de l'équipe.

Après une courte introduction sur les objets agrégés, sont présentées les deux parties principales de mon travail ; les formats d'agrégations et la sûreté de fonctionnement des systèmes fondés sur les objets agrégés. Des parties de mon travail ont été mises en annexes car elles étaient trop techniques (comme l'annexe F), trop théoriques (comme l'annexe C) ou trop longues pour être intégrées dans le corps du rapport (comme les annexes B et D).

Abstract

This report shows my work during my three-month internship in the team ACES (Ambient Computing and Embedding Systems) in the INRIA Rennes — Bretagne Atlantique research center. This internship has been supervised by Michel Banâtre, head of the team ACES, and Fabien Allard, engineer in the start-up company SenseYou.

After a short introduction, the two main parts of my work are presented: aggregating formats and dependability of aggregates based system. Some informations have been put in appendix. They are also part of my work but are either too much technical (like annex F), too much theoretical (like annex C) or too long to be in one of the main parts (like appendices B or D).

1 Introduction

Checking for integrity of a set of objects is often needed in various activities, both in the real world and in the information society. The basic principle is to verify that a set of objects, parts, components, people remain same along some activity or process, or remains consistent against a given property (such as a part count).

In the real world, it is a common step in logistic: objects to be transported are usually checked by the sender (for their conformance to the recipient expectation), and at arrival by the recipient. When a school get a group of children to a museum, people responsible for the children will regularly check that no one is missing. Yet another common example is to check for our personal belongings when leaving a place, to avoid oversights. While important, these verification are tedious, vulnerable to human errors, and often forgotten.

Because of these vulnerabilities, problems arise: passengers forget bags in airplanes, for instance.

While there are very few automatic solutions to improve the situation in the real world, integrity checking in the computing world is a basic and widely used mechanism: magnetic and optical storage devices, network communications are all using checksums and error checking code to detect information corruption, to name a few.

The emergence of Ubiquitous computing and the rapid penetration of RFID (Radio Frequency IDentification) devices enables similar integrity checking solutions to work for physical objects.

1.1 Aggregated objects and basic concepts

1.1.1 Basic aggregated objects

More precisely, proposed system is able to check integrity of **(physical) aggregated objects** (or **(physical) coupled objects**). Basic aggregated objects are sets of mobile and/or physically independent physical objects, called **fragments**.

Physical objects can be aggregated by an **aggregating system** using an **aggregating algorithm**.

Then, integrity of the resulting aggregated object can be verified by a **verifying system** using a **verifying algorithm**, in some determined areas, called **verifying areas**. More precisely, in basic cases, if fragments forming an aggregated object are put together in such an area, a treatment is done: for instance opening a door. But if the fragments in a verifying area do not form all together an aggregated object (because one fragment is missing or some fragments belong to different aggregated objects), an exception is thrown; this exception can sett off an alarm.

Anyway, outside verifying areas, fragments can be moved as you want.

A direct application of this basic mechanism is UbiCheck. UbiCheck helps travelers not forgetting one of their items, or mistakenly exchanging a similar item with someone else. At airport entrance, each passenger is aggregated with all its items (hand baggages, billfold and jacket for instance). Then, integrity checks are performed after X-rays, before entering airplane and after exiting airplane.

1.1.2 Analogy with packet switching

One can compare this mechanism with packet switching. When a sender wants to send a file to a receiver through a network, it cuts the file into packets and send these independent packets through the network. The receiver receives the packets, put them together and verify the result is integral.

Packets correspond to fragments and the file is like an aggregated object. Verifying areas are the receiver and sometimes also routers which can check the integrity of the file. Outside this area, packets are completely independent and can even use different ways to go from the sender to the receiver.

Another important special feature is that information needed to reconstruct the file is in packets themselves. This is also the case with the used implementation of aggregated objects: information needed to know which fragments form which aggregated objects are stored on fragments themselves. There is no external database.

1.1.3 Tree aggregated objects

There are also more complex aggregated objects: tree aggregated objects. Tree aggregated objects can contain an aggregated object as a fragment.

Example 1.1.1. *The tree aggregated object $\{\{\{a, b\}, \{c, d, e\}\}, f\}$ corresponds to the tree of the figure 1. The leaves are real fragments, also called fragments leaves. The (internal) nodes h_i are both aggregated objects and fragments, except h_4 which is only an aggregated object.*

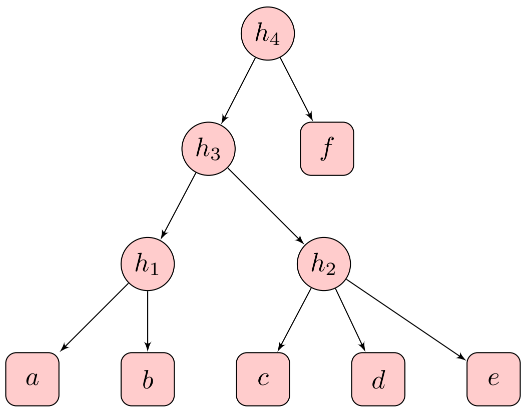


Figure 1: An example of tree

The goal of a tree aggregating algorithm is to store data in fragments leaves such that it becomes possible to find the maximum subtrees (maximum for the inclusion) which contains a given set of leaves. The way these data are stored are called an **aggregating format**.

Example 1.1.2. *Suppose the only created aggregated object is the one depicted in the figure 1.*

- *If fragments a, b, c and d are in the verifying area, the verifying algorithm shall find these fragments form the aggregated object of root h_3.*
- *If fragments a, b, c and e are in the verifying area, the verifying algorithm shall find a and b form the aggregated object of root h_1 and c, e do not form an aggregated object.*

1.2 Real implementation

Actually, leaves of an aggregated object are real life objects and thus cannot embed information, as is. The used solution is to put a RFID tag on each fragment leaf. A **RFID tag** is a small electronic device with a chip and an antenna. They can be inlays (adhesive tags), cards (badges) or directly embedded in a physical object (for instance, casted in a bike frame).

RFID tags can be read and modified by a **tag interrogator** over the air. Actually a tag interrogator is often connected to a **controller** on which runs an aggregating algorithm or a verifying algorithm. A controller and a tag interrogator form an aggregating and/or verifying system.

Tags often have two kinds of memory data:

- a read-only (factory programmed and unique) **identifier** or **id**, which identifies the fragment.
- a writable **user memory**, often simply called **memory**, which contain aggregation data: data used to know to which aggregated object the fragment belongs.

Since tags are attached to fragments leaves and contain aggregating data, fragments leaves are often also called tags.

1.3 Example of uses

This section illustrates how aggregation can solve several concrete issues. Two examples are to be depicted: UbiQuitus and UbiPark. In both projects, object aggregation principles enable risks reduction and security.

1.3.1 UbiQuitus

UbiQuitus is a standalone system aiming at securing safety zones by mean of coupling operations. The main risk in such zones is that dangerous objects, or set of objects, that can be brought into it. UbiQuitus does not provide a full protection as bombs will not be detected, thus, usual security controls when entering the area should not be interrupted. However, the system provides help to the security controls in the area: it enables to check if somebody is carrying all the equipment he brought to the area or if something is missing. This way, people cannot stash potentially dangerous equipment in the area without triggering an alert during a checking operation. Those scenarios are depicted in the figure 2a.

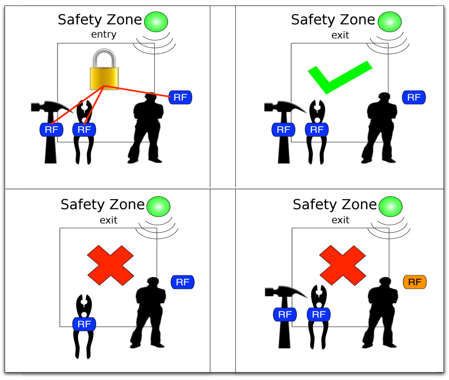
Checking operation can be processed at each exit from the area, or at any time, using handhold checking devices.

1.3.2 UbiPark

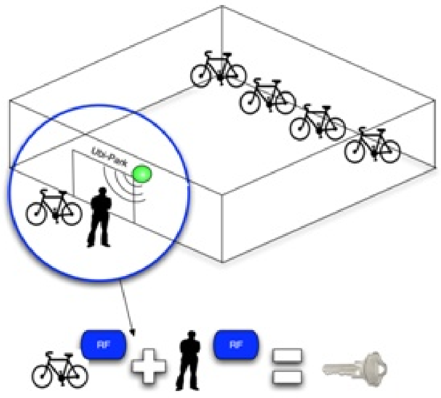
UbiPark is a standalone system aiming at providing access control and monitoring a bike shed (see figure 2b). It grants access to any user that is coupled with his bike. The key enabling to access the shed is the coupled object.

People can only enter the shed with their bike, or alone if their bike is already inside it, see the figure 3b.

The same way, they cannot exit with somebody else's bike: they can only exit alone or with their bike, see the figure 3a.

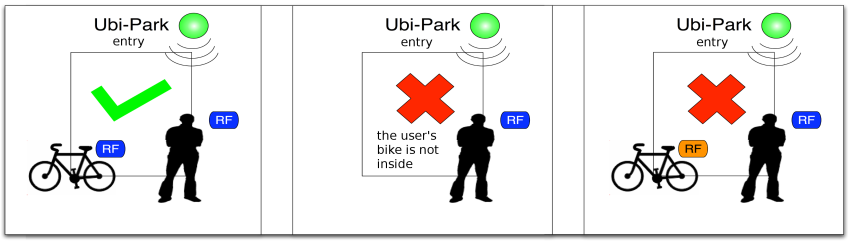


(a) UbiQuitus

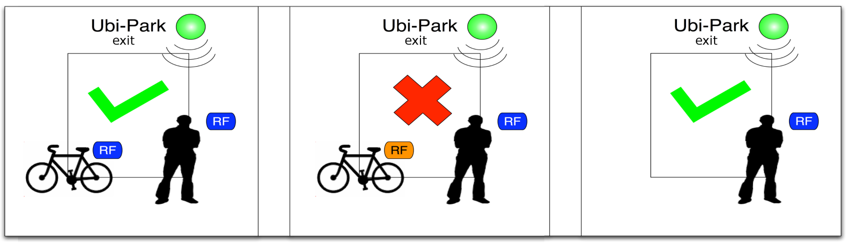


(b) UbiPark

Figure 2: UbiQuitus and UbiPark



(a) Entrance



(b) Exit

Figure 3: UbiPark (entrance and exit)

In this application, users are equipped with a unique tag and their bike should at least embed one tag. The tree depicted on figure 4 gives an example of an UbiPark's coupled object.

Different parts of the bike (front wheel, back wheel, saddle, handlebars) are tagged and aggregated together. Then, this aggregate is coupled with the owner's tag. The label "bike" and the root of the tree are the two aggregated objects and they do not correspond to any tag.

One advantage using such a tree aggregated object is that it is possible to verify the bike (to ensure nobody has stolen a part of the bike) and to verify the coupled object "bike + owner" (which is the key of the shed).

1.4 Global architecture

A typical application can be segmented in 5 main layers shown in figure 5.

New data are propagated using an event-driven model, from the first to the last layer. The raw data that are physically written on the tag memory (layer 1) are read by the raw data providing layer (layer 2). Layer 3 is in charge of filtering, parsing, decrypting and authenticating those data. It builds a virtual context upon those data and notifies the upper layer (layer 4: aggregation service) of any of its

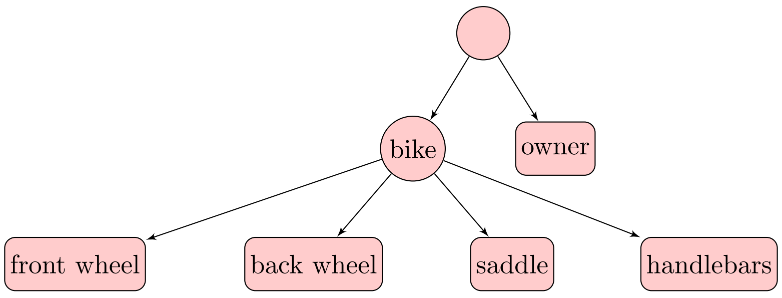


Figure 4: Example of UbiPark's coupled object

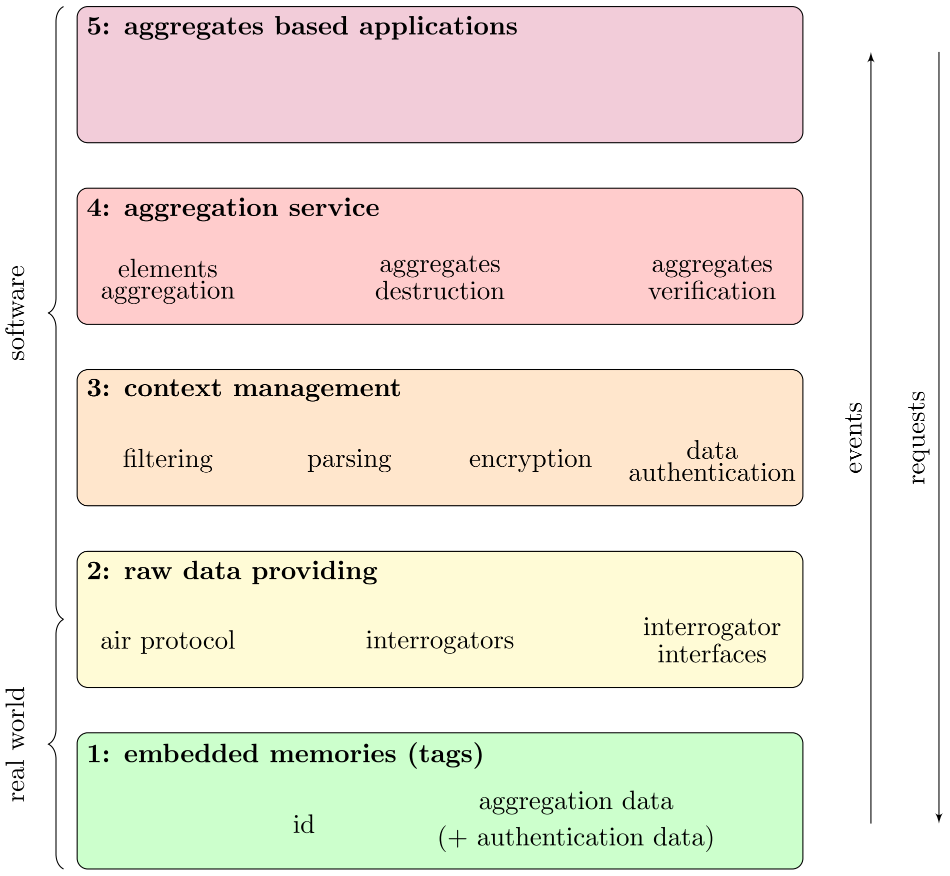


Figure 5: Global architecture layers

changes. Every time a context notification is sent, the aggregation service (layer 4) searches for aggregates structures in the context and notifies aggregates based applications (layer 5) of any structure change. Each level can also ask for lower level operations to the previous layers. As an example, a checking algorithm from layer 4 can ask for tag data authentication to layer 3 before notifying layer 5.

An UML model of this global architecture has been created.

Part I

Aggregating formats

This first part approximately corresponds to the first part of my internship: the goal was to find tree aggregating formats and to try to generalize them. Before this work, the only aggregating format was the original format (section B.1).

The first section introduces common concepts and remarks on aggregating formats. Some of these concepts (in particular hash tree — section 2.1 — and first verifying algorithm) were already used. Other concepts have been introduced or modified for tree aggregating formats. The second section describes the last discovered tree aggregating format and the idea common to all proposed tree format. The third section introduces centered multiple parent tree aggregating format, a generalization of tree aggregating formats.

2 Aggregating format generalities

This section introduces common concepts and remarks for all aggregating formats.

2.1 Hash tree and tag memory

With all tree aggregating formats[1], tree leaves are tags and nodes do not contain any memory and correspond to aggregated objects. Each leaf or node has an **id** (which is a natural number). For leaves, id is the tag id which shall be unique. For nodes, it is the hash value (or digest) of the concatenation of the children ids. Node ids are also called **digests**.

So a complete tree aggregated object forms an hash tree.

Intuitively the id of the root node approximately defines the tree if the used hash function is not too "simple". More precise properties can be found in section 3.3.

Example 2.1.1. *The figure 6 is an example of a tree aggregated object.*

In particular, h_1 is the hash value of the concatenation of a and b; and h_3 is the hash value of the concatenation of h_1 and h_3

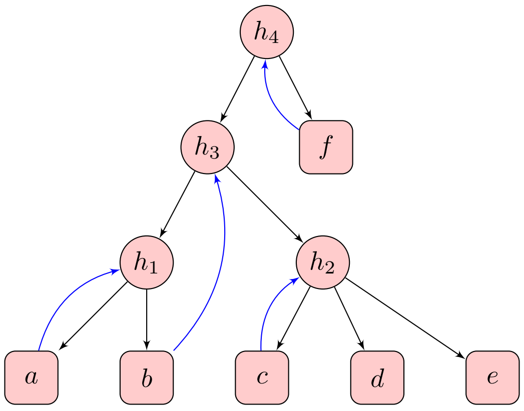


Figure 6: An example of hash tree

Furthermore, in all aggregating formats described in this report, each tag stores node ids or parts of node id in its user memory. In figures, colored arrows indicate which node ids are stored in tags. For example, in the figure 6, the tag a stores the id h_1.

It may seems possible to store just zero or one node id in each tag such that for all subtree, each node id is stored in one tag. Figure 6 gives an example of this aggregating format. It can work, because a node

[1]This is also true with a more general aggregating format: the multiple parents trees. But explaining it in this case requires introducing several definitions. That is why, this subsection is limited to tree aggregated formats.

id defines the subtree whose the root is the considered node. But it has two main disadvantages. Firstly the verification is very slow (one needs to check all possible trees[2] to find which of them corresponds to the given digest. Secondly there is absolutely no security since it is very easy to add a tag to such an aggregated object.

Hence more complex aggregating format are used.

2.2 Two remarks

This subsection gives two common remarks for all aggregating formats. The first one is not very important but the second one has a big impact on security.

2.2.1 Children, children's order and left/right children

In this paper, each node has two or more children. A node with one child is replaced by a single node (see figure 7) because, otherwise it would be impossible to store some trees in tags (a tree could have an unbounded number of nodes and only one leaf).

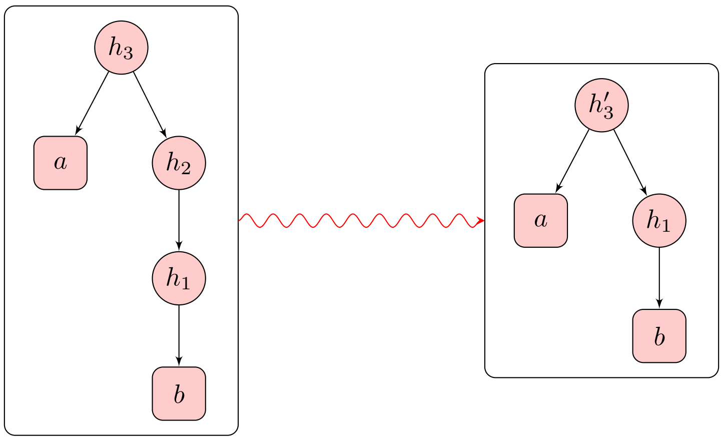


Figure 7: Node with only one child are removed.

According to the first paragraph of this section, the id of a node with three children whose ids are a, b and c depends on the order (in the tree) of the children, since it is: $H(a, b, c)$. Three solutions are possible:

- verifying algorithms may test all possible orders.
- order of children may be forced to be the increasing order of the ids.
- aggregating algorithms may store the order.

The last solution depends a lot on aggregating algorithm, may not work with all algorithms and will not be discussed in this report. In order to simplify further explanations, the children's order is supposed to be the increasing order of ids. That means the children order is not stored.

The child with the lowest id is called a **left child**. The child with the highest id is called a **right child**. The other children may be either left or right children (it does not matter as soon as it is fixed).

2.2.2 Concatenation and size issues

Warning 2.2.1. *Naive implementation of hashing in tree aggregating algorithms (section 2.1: hash of the concatenation of the children ids) may lead to critical security flaws. Some precautions shall be taken.*

[2] "Possible trees" are possible organizations of the given tags in a tree: not only the tree's structure (if all tags are children of the same root or if tags are organized in a binary complete trees, ...) but also the place of each tag in the tree or which leaf corresponds to which tag.

Suppose an attacker can create tags with several short ids (several bits long) but not with normal ids (about 64 bits long)[3]. To simplify, suppose also that children order can be chosen (and is not the increasing order).

It may be so likely a lot of normal ids may be cut in smaller ids (i.e. the concatenation of the small ids gives the original id) for which it is feasible to create a tag. It becomes so possible to replace a tag by several tags even if the original tags were unclonable and even if the used hash function is very secure.

Example 2.2.2. *Suppose you can create tags with binary ids* 100*,* 101 *and* 1000 *and suppose you have lost (or you want to clone) a tag with id* 10010001011000101*.*

You can replace your "lost" tag by tags with id 100*,* 1000*,* 101*,* 1000 *and* 101 *(in this order).*

A first workaround is to ensure that all ids have the same length. All controllers must check that point otherwise attacks are possible.

However since there are two kinds of id (tag id and digest), ids may not always have the same size. Instead of hashing the concatenation of all ids $h_1, \ldots, h_n$, it is possible to hash the following data:

n	$s(h_1)$	$\cdots$	$s(h_n)$	h_1	$\cdots$	h_n

where $s(h_i)$ is the size of h_i (in bytes or in bits depending of the application).

The sizes of n and $s(h_i)$ fields must be fixed.

Suggestion 2.2.3. *If $s(h_i)$ is in bits, size of n and $s(h_i)$ may be two bytes (16 bits). In practice, it is almost always sufficient.*

In order to simplify further proofs, some notations are added. The real hash function (whose input is a bit string) is $\tilde{H}$ and the function which transforms a list of ids into an input of $\tilde{H}$ (concatenation of same length ids or more complex mechanism described above) is f:

$$H(h_1, \ldots, H_n) = \tilde{H}(f(h_1, \ldots, h_n))$$

Remark 2.2.4. *Actually, f may be any bijection from a subset of lists of bit strings to the set of bit strings. And the implementation of f shall verify its inputs are in the chosen subset (and shall throw an exception if it is not the case).*

2.3 Aggregating and verifying algorithms

Aggregating and verifying algorithms of all aggregating formats are very near each others. This subsection introduces a generic aggregating algorithm and a generic verifying algorithm.

2.3.1 Aggregating algorithms

An aggregating algorithm takes a set of trees (i.e. tree aggregated object or tag) and **aggregates** them i.e. modify the tags of these trees to create a new aggregated object whose the root is a new virtual aggregated object and the children of the root are the input trees. It obviously enables to create any tree aggregated object (recursively). An example is given in the figure 8.

All aggregating formats need a verifying algorithms. A generic verifying algorithm is presented in this paragraph. Another is described in appendix A.

2.3.2 Verifying algorithms

Here is described one generic verifying algorithm (called for historical reason: the second generic verifying algorithm). In annex A, another one (called the first generic verifying algorithm) is described.

This second generic verifying algorithm (algorithm 2.3.2.1) takes the set of tags seen by the tag interrogator. And it returns the maximum trees containing these tags.

[3]This assumption is not weird. Actually, to prevent creating a tag with a given ids, it is possible to enforce tags id are public key of a zero-knowledge identity proof, like Fiat-Shamir. The private key is only known by the tag and the tag can prove it knows its thanks to the identity proof. Unfortunately, it is possible to compute the private key related to some short Fiat Shamir public keys (when these keys are square number).

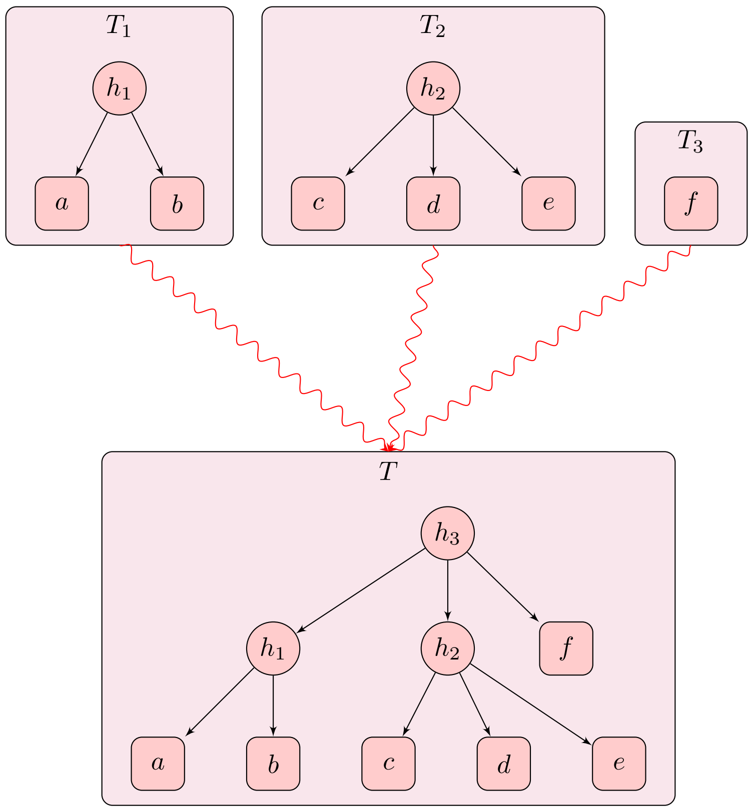


Figure 8: Aggregation of three trees T_1, T_2 and T_3 — T is the resulting aggregated object.

To use this algorithm with a given aggregating format, it shall be indicated how the parent digest of a tree T can be computed thanks to the data in the tags of T.

Example 2.3.1. *Suppose the aggregated object of the figure 6 has been created by a good aggregating algorithms (the colored arrows are not the one written on the figure).*

Suppose the tag interrogator sees tags a, b, c, d, e and another non-aggregated tag g but not the tag f. $S = \{a, b, c, d, e\}$.

When check2(S) is executed:

1. *it first computes the parents of a, b, c, d, e and g which are h_1, h_1, h_2, h_2, h_2 and null (no parent) respectively. So:*
$$E = \{(a, h_1), (b, h_1), (c, h_2), (d, h_2), (e, h_2), (g, null)\}$$

2. *it sees that a and b have the same parent, verify that the tree $\{a, b\}$ has a root id equal to h_1 (i.e. $H(a, b) = h_1$).*

3. *it computes the parent id of this tree: h_3.*

4. *it replaces (a, h_1) and (b, h_1) by $(\{a, b\}, h_3)$ in E. So:*
$$E = \{(\{a, b\}, h_3), (c, h_2), (d, h_2), (e, h_2), (g, null)\}$$

5. *it repeats the steps 2 to 4 with c, d and e. E becomes:*
$$E = \{(\{a, b\}, h_3), (\{c, d, e\}, h_3), (g, null)\}$$

6. *it repeats the steps 2 to 4 with $\{a, b\}$ and $\{c, d, e\}$. E becomes:*
$$E = \{(\{\{a, b\}, \{c, d, e\}\}, h_4), (g, null)\}$$

Algorithm 2.3.2.1 Second generic verifying algorithm: $check2(S)$

```
Require: $S$ is the set of tags seen by the tag interrogator.
  $E \leftarrow$ the set of ordered pairs $(u, h)$ where $u$ is a tag of $S$ and $h$ is the parent id of $u$.
Ensure: During this algorithm, $E$ is always a set of ordered pairs $(T, h)$ where $T$ is an aggregated object
  and $h$ is the parent id of $T$ (or $null$ if the parent id does not exist). Furthermore, each tag of $S$ is in
  one of the aggregated objects contained in $E$.
  while two ordered pair $((T_1, h)$ and $(T_2, h))$ of $E$ have the same second element $h \neq null$ do
    Let $(T_1, h), \ldots, (T_n, h)$ all the ordered pairs of $E$ whose the second element is $h$.
    $B \leftarrow \{T_1, \ldots, T_n\}$
    if a subset $B'$ of $B$ can be an aggregated object (i.e. digest is correct) then
      Remove elements of $E$ whose the first element is in $B'$.
      $h' \leftarrow$ parent of the aggregated object $B'$
      $E \leftarrow E \cup \{B', h'\}$
    else
      Replace $h$ by $null$ in the pair $(T_1, h), \ldots, (T_n, h)$ of $E$.
    end if
  end while
  return The set of first elements of ordered pairs of $E$
```

7. *it returns the following set (with one aggregated object and one tag):*

$$\{\{\{a, b\}, \{c, d, e\}\}, g\}$$

3 Tree aggregation formats

There are different aggregating formats:

- **original format** is the first used format. This format cannot represent all tree aggregated object. This format is described in the annex B.1.
- **first tree format** supports all trees with a limited height for the basic version and without any limitation for the enhanced version.
- **second tree format** can support all trees but for performance purpose, a limited height version is used.
- **third tree format** supports all trees and should be the preferred aggregating format.
- **centered multiple parents tree (CMPT) format** can support more general structure than trees but is less secure. The CMPT format is not described in this section but in the next one (section 4).

The three tree formats use the same idea. This idea is described in section 3.1.

The third tree format is almost always the best choice, except when some security properties of the first tree format (section 3.3.4) are required. That is why the only described tree format in this section is the third tree format. Other tree formats are described in the annex B.

3.1 Idea of tree formats

The three tree formats are based on the same idea.

Let L be the size (in bits) of node ids or digests. Each tag memory is supposed to have at least $2L$ bits for storing node ids.

Let call **free bits** of a tag, the set of bits which are not used to store aggregating data. When a tag is alone, it has at least $2L$ free bits. Let call **free bits** of an aggregated object, the union of the free bits of tags of this aggregated object.

All proposed tree formats are created with the algorithm 3.1.0.2. Each aggregated object or tag has always at least $2L$ free bits. This invariant proves the algorithm can always be executed.

Algorithm 3.1.0.2 Generic tree aggregating algorithm: *aggregate*(E)

```
Require: $E$ is a set of trees to be aggregated (as explained in section 2.3)
Require: each tree of $E$ is a correct aggregated object. In particular each tree has at least $2L$ free bits.
Ensure: return aggregated object whose children of the root are the elements of $E$
  $L \leftarrow$ the sorted (by id) list with the elements of $E$
  $L_{id} \leftarrow$ the related list of ids
  $h \leftarrow$ the digest $H(L_{id})$ (see section 2.1)
  Write the digest $h$ in $L$ free bits of each tree of $E$.
  return the aggregated object which corresponds to $L$
```

Obviously the free bits used to write the digest h must be smartly chosen such that it will be possible to easily retrieve the digest of a parent of any aggregated object. Indeed the generic verifying algorithms need to retrieve this digest (see section 2.3 — this is not sufficient for the first generic verifying algorithm A.1.0.1).

Remark 3.1.1. *If aggregated objects are only binary trees, there are always exactly $2L$ free bits. Otherwise, there may have more free bits. It is possible to use these free bits to introduce redundancy or to represent more general (than tree) aggregated objects) as explained at the end of the section 3.2.*

Remark 3.1.2. *For the second and the third tree formats, the $2L$ bits are distributed this way: the first L bits contain a complete node id called the **first digest** (or the **left digest**) and the last L bits contain a complete node id called the **second digest** (or the **right digest**).*

*A digest of a tag is called **free** if it is not used. Generally, the following convention is used: if a digest is 0, it means it is not used. Unfortunately, it can happen, a real digest can be 0. But since it is quite* *impossible*[4]*, it may be used. Otherwise, a bit (which can be stored in the tag or only in the aggregating controller*[5]*) may be used to indicate if a tag digest is used or not.*

3.2 Third tree format

The third tree format is simpler than the two other tree formats. It should be used except if spread properties of the first tree format are required (see remark 3.3.4).

The first digest contains the father's id. When two aggregated objects are aggregated, the leftmost second free digest is used in each aggregated object.

The algorithm 3.2.0.3 is the aggregating algorithm for this format.

Algorithm 3.2.0.3 Third tree aggregating algorithm: $aggregate(E)$

```
Require: $E$ is a set of trees to be aggregated (as explained in section 2.3)
Require: each tree of $E$ is a correct aggregated object (with third tree format).
Ensure: return aggregated object whose children of the root are the elements of $E$
  $L \leftarrow$ the sorted (by id) list with the elements of $E$
  $L_{id} \leftarrow$ the related list of ids
  $h \leftarrow$ the digest $H(L_{id})$ (see section 2.1)
  for all tree $T \in E$ do
    if $T$ is a leaf then
      Write $h$ in first digest of tag $T$.
    else
      Let number leaves of $T$ in the order given by the tree (the order of the leaves when the tree is drawn): $1, \ldots, k$.
      $i_0 \leftarrow$ the lowest $i$ such that the second digest of the tag $i$ is free
      Write $h$ in the second digest of tag $i_0$.
    end if
  end for
  return the aggregated object which corresponds to $L$
```

The verifying algorithm 2.3.2.1 and A.1.0.1 can be used. A way to find the parent of any aggregated object is to store in the controller (or in the verifying system) which digests are used by all found aggregated objects. The leftmost free digest of the aggregated object is the digest of the parent.

To improve the speed, the verifying algorithm can store the list of leaves whose the second digest is free, for each found aggregated object. The first element of the list contains the digest of the parent. A linked list may be a good idea for the implementation, because the two main operations are concatenation and popping first element.

Remark 3.2.1. *An advantage of this format is the fact that only two tags need to be changed when two objects are aggregated.*

In addition, each aggregation changes only L free tag bits: one of the two digests. So it is possible to use tags where digests are write-once (tags where memory is write-once per block and contains two blocks of L bits).

Generalization

In the previous aggregating format, there are unused tag digests (at least two, the worst case is binary tree — with other trees, there are more than two unused tag digests). These digests can be used to:

- save the grandfather in a tree of height two for example. So the format become the original aggregating format. One advantage is that only one tag digest must be read to find the grandfather.

[4]With a supposed perfect hash function whose the output has n bits, the probability of such an event is 2^{-n}.

[5]If the bit is only stored in the aggregating controller, it is recommended not to use several different controllers to build an aggregated object — creating an aggregated often requires many steps when the height of the tree is greater or equal than 2. Anyway, if it is not possible to say if a digest is used or not, it is not possible to say if a set of tags composed a whole aggregated object or just a part of a bigger aggregated object (a subtree).

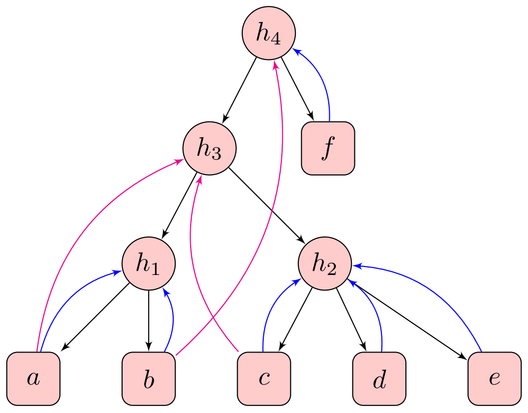


Figure 9: Third tree format example

- save another father or parent. There are many ways to do this, depending on the shape of the final MPT. It may be useful[6] to store, in each tag, the number of edges between the tag and the node whose the id is stored in the second hash (this number is called relative depth in the second tree format — see section B.3). This is a light version of multiple parents trees (MPT — see section 4). Figure 10 gives an example but there are many other possibilities.

If tag memory size is big enough, other digests may be stored and more MPTs may be supported. Only few modifications need to be performed to the previous algorithms.

If some tags do not have any memory except id, it can be possible to do aggregated objects. Memoryless tags have often a determined place in tree for practical applications (for example, in UbiPark, the owner tag may be memoryless — section 1.3.2) and so it is often easy to verify such an aggregated object (even if in general cases it is complex).

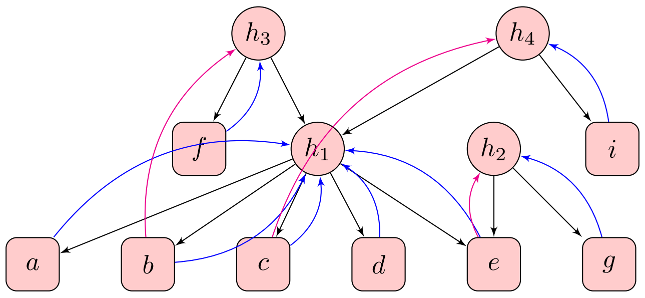


Figure 10: Generalized third tree format example

3.3 Security of aggregating and verifying algorithms and hashing functions

This subsection focuses on the security of tree aggregating format. It also applies to original format, if the created aggregated objects are only those allowed by this format (and not all trees).

All presented tree aggregating formats (in section 2) use an hash tree: id of each node is the hash value of the concatenation (or more secure methods described in the next section) of ids of the children. It may seem id of internal node could be any unique number (for example a random number or a timestamp). This method can reduce the memory usage of the tag but also reduces security.

Suppose the used hash function $\tilde{H}$ provides second preimage resistance i.e. given any input m_1, it is difficult to find another input m_2 such that: $\tilde{H}(m_1) = \tilde{H}(m_2)$. Let call “difficult” an operation that lead to find a second preimage.

[6]But not mandatory. For example, if a bike, composed of three tags, has three owners. The digest corresponding to each owner can be stored in a second digest of a different tag.

Proposition 3.3.1. *Aggregated objects (with a tree aggregating format or the original aggregating format) verify the following properties:*

1. *adding a tag (or a subtree) to a read-only tree aggregated object*[7] *(as a new leaf of the tree) is difficult.*
2. *replacing a lost tag (or subtree) in a read-only tree aggregated object by one or more tags, such that the resulting tree is a good aggregated object, is difficult except by finding the original id of the tag (or subtree).*
3. *replacing a tag in a read-only tree aggregated object by one or more tags, such that resulting tree is a good aggregated object, is difficult, if tag ids are unique (in the following meaning: it is not possible to create a tag with a given id).*

Proof. Let h be the id of the father u of the added (for the first point) or replaced (for the two other points) tag or subtree. Let $h_1 < \cdots < h_n$ be the ids of the children of u (in the increasing order).

$$h = H(h_1, \ldots, h_n)$$

For the first point, let h' be the id of the added tag or subtree (see the figure 11). Suppose, without loss of generality, that $h' \geq h_n$. Since the aggregated object is read-only and considering the aggregating formats, the id of u cannot be changed (because it is stored in one read-only tag) and so:

$$h = \tilde{H}(f(h_1, \ldots, h_n)) = \tilde{H}(f(h_1, \ldots, h_n, h'))$$

Size of $f(h_1, \ldots, h_n, h')$ and $f(h_1, \ldots, h_n)$ are different and so $f(h_1, \ldots, h_n, h') \neq f(h_1, \ldots, h_n)$. Therefore $f(h_1, \ldots, h_n, h')$ is a second preimage of $\tilde{H}(f(h_1, \ldots, h_n))$ and so computing h' is difficult.

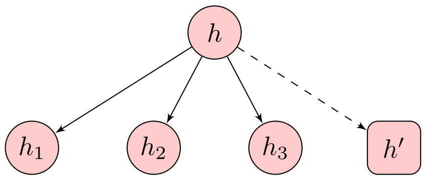


Figure 11: Difficulty to add a tag h' to a read-only aggregated object.

For the second point, let h' be the id of the replaced tag or subtree. Suppose, without loss of generality, that the lost tag has id h_n and that $h_i \leq h' \leq h_{i+1}$ ($h_{-1} = 0$ and $h_{n+1} = \infty$).

$$h = \tilde{H}(f(h_1, \ldots, h_n)) = \tilde{H}(f(h_1, \ldots, h_i, h', h_{i+1}, \ldots, h_{n-1}))$$

Let a be $a = f(h_1, \ldots, h_n)$ and b be $b = f(h_1, \ldots, h_i, h', h_{i+1}, \ldots, h_{n-1})$. Since it is supposed that $h' \neq h_n$, it is clear that $a \neq b$. So b is a second preimage of $\tilde{H}(a)$. Notice that a is not known but only a part of a and a weaker property than second preimage resistance may be sufficient to ensure security of this second point.

For the last point, keep the same notations as for the second point. The only problem is when $h' = h_n$. Suppose that h_n was a tag id and h' a subtree's digest (if h_n is a subtree or if h' is a tag's id, the proof is nearly the same). Let $h'_1 < \cdots < h'_{n'}$ be the ids of the children of h':

$$h' = \tilde{H}(f(h'_1, \ldots, h'_{n'}))$$

$f(h'_1, \ldots, h'_{n'})$ is a preimage of h'. Since preimage resistance is clearly a stronger property than second preimage resistance, the proof is finished. □

Warning 3.3.2. *For the second property, if the size of the id is small, an attacker just need to try all possible ids to find the original id. If tag ids are consecutive (which is the case for a lot of commercial tags), finding the lost id is really easy.*

Therefore, if the second property is needed (and if ids are not unique), ids shall be large (really) random numbers.

[7] A read-only tree aggregated object is an aggregated object whose all tags are read-only: their user memories (and their ids) cannot be changed.

These properties are not verified when digests are replaced by unique numbers or when the hash function is not second preimage resistant. If other cryptographic mechanisms prevent anyone from creating tags or from parsing tag data, random number or non second preimage resistant hash functions can be used, as soon as collision probability is not too high (because if it is not the case, several internal nodes could have the same id and with random number, verifying an aggregated object would become impossible whereas with non second preimage resistant hash function, aggregating or verifying algorithm would need to test all possible trees to find the good one). If uniqueness of ids cannot be ensured, and if they are random (or like a random number — this is in the case when ids are hash values), the number of possible ids shall be greater than the square of the number of used tags (or so) because of the birthday problem (see wikipedia article [45]). However the resulting id size is often still much lower than the digest size of a second preimage resistant hash function.

Remark 3.3.3. *It is possible to secure some tag data (for example an expiring date) thanks to aggregating format: just concatenate these extra data to id of tag and hash this new data instead of hashing the id, if extra data have always the same size. Otherwise concatenation's issues (section 2.2.2) shall be taken into account.*

Remark 3.3.4. *The first tree format has a small advantage over the two other tree formats: the bits of the digests are spread on a lot of tags (most cases on all tags). So even if a tag is not a read-only tag, it may be difficult to complete an aggregated object for example (since changing data of this tag can only change a part of the digest).*

4 Aggregation format for multiple parents trees

This section introduces the CMPT aggregating format. CMPTs are generalization of trees.

In UbiPark, they could be used to enable a bike to have more than one owner. However, in most cases, generalized third tree format (section 3.2) is sufficient to do so.

4.1 Multiple parents tree (MPT) and centered multiple parents tree (CMPT)

This subsection gives the intuition of two novel notions: multiple parents tree (MPT) and centered MPT (CMPT). All names and definitions are invented. A more precise formalization can be found in annex C. But these formal definitions are not required to understand this section.

A MPT is intuitively a tree where each internal node can have more than one parent. Vocabulary of trees (root, node, leaf, parent, generation, ...) applies to MPT.

A CMPT is a particular MPT, where only one node of each generation can have more than one parent.

A **binary** CMPT is a CMPT where each node has 2 children.

Binary CMPTs can also be recursively defined as follow:

- a graph with only one vertex and without any edge is a CMPT.
- if $(G, G_1, \ldots, G_n)$ are $n+1$ CMPTs and $u, u_1, \ldots, u_n$ are minimum generation nodes in $G, G_1, \ldots, G_n$ respectively, the graph containing $G, G_1, \ldots, G_n$, with n new nodes $w_1, \ldots, w_n$ and $2n$ new edges:

  $$(w_1, u_1), (w_1, u), \ldots, (w_n, u_n), (w_n, u)$$

  is a CMPT.

It is possible to extend this definition to CMPTs by adding the possibility a w_i (in the second point) can have more than two children as soon as one of these children is u. But notations become ugly.

The figure 12 shows this recursive construction. u and G can be considered as in the **center** of the built CMPT.

4.2 Bloom filter

CMPT aggregating format uses Bloom filters.

A Bloom filter is a probabilistic data structure which represents a set (here it is a subset of $\mathbb{N}$). Enabled operations are:

- insert an element in the set
- test whether a given element is in a set (there can be false-positives but not false-negatives)

More precisely a Bloom filter is an array of m bits (all set to 0) associated with k hash functions $(h_1, \ldots, h_k)$ from $\mathbb{N}$ to $\{1, \ldots, m\}$.

- to add the element $a \in \mathbb{N}$ to the Bloom filter, set to 1 bits at positions $h_1(a), \ldots, h_k(a)$.
- to test whether a is in the Bloom filter, check whether bits at positions $h_1(a), \ldots, h_k(a)$ are set to 1.

After n insertions, the probability of false-positives is approximatively (see [46] — for a more precise approximation see [12]) :

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$$

and the minimum is obtained for $k \approx \frac{m}{n} \ln 2$.

The figure 13 shows the probability of false-positives in relation to the number of insertions n.

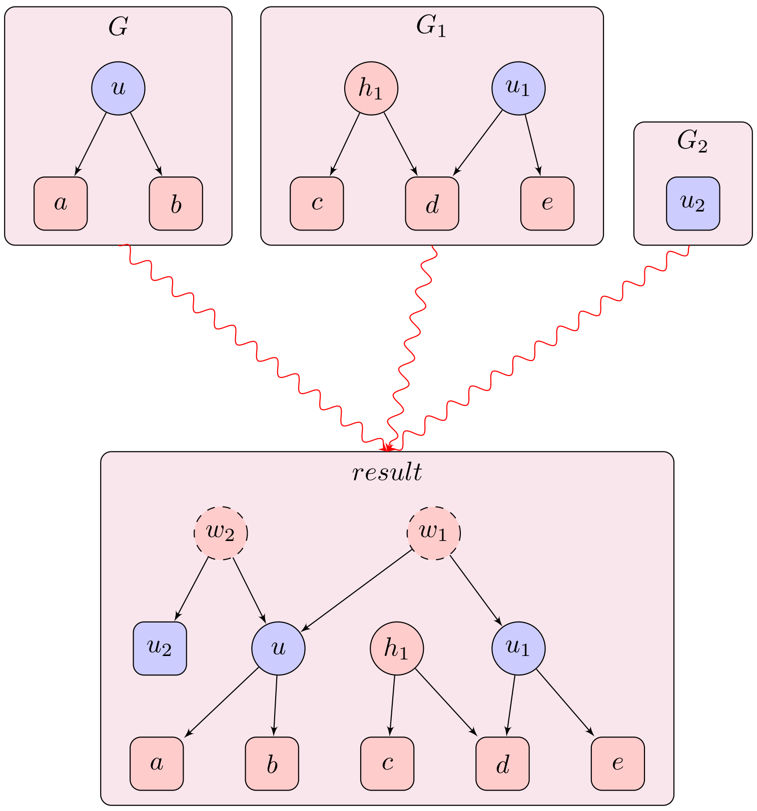


Figure 12: Illustration of the recursive construction of CMPTs

4.3 Aggregating format and algorithms for CMPTs

In this subsection, the aggregating format for CMPT is described. As for tree aggregating format, tags are the leaves and can store data whereas nodes do not have any memory.

The algorithm can support any CMPT in the following meaning: there is a verifying algorithm which can find the maximum subtrees whose the leaves are in a given set of tags (or leaves). Actually this is quite the same thing as for tree aggregating format (see section 2.3).

4.3.1 Aggregating format and algorithms

Exactly as with tree aggregating format, the id of a node is the hash value of the concatenation of its children id (or better mechanism see section 2.2.2).

Each tag user memory contains one digest and one bloom filter (see annex 4.2). The parameters of the Bloom filter are discussed in the section 4.3.2.

The basic idea of this aggregating format is very near the one of the third tree format (see section 3.2).

In the two first subsections, CMPTs are supposed binary. The last subsection generalizes the results to all CMPTs.

Aggregating algorithm (binary CMPT case) The following invariant is verified for each aggregated object (CMPT): for each subtree (of the CMPT) whose the root has the smallest generation, there are at least one tag whose the bloom filter is empty and there are at least one tag whose the digest is free (i.e. not used for aggregating data).

The building of an aggregated object is done thanks to the recursive construction of CMPTs. The algorithm 4.3.1.1 is the aggregating algorithm.

The invariant is verified because in the new CMPT, there are n subtrees whose the root has the smallest generation: the subtrees of root w_1, ... and w_n; and in the subtree of root w_i, there is an empty bloom filter in the sub-subtree of root u_i and a free digest in the sub-subtree of root u.

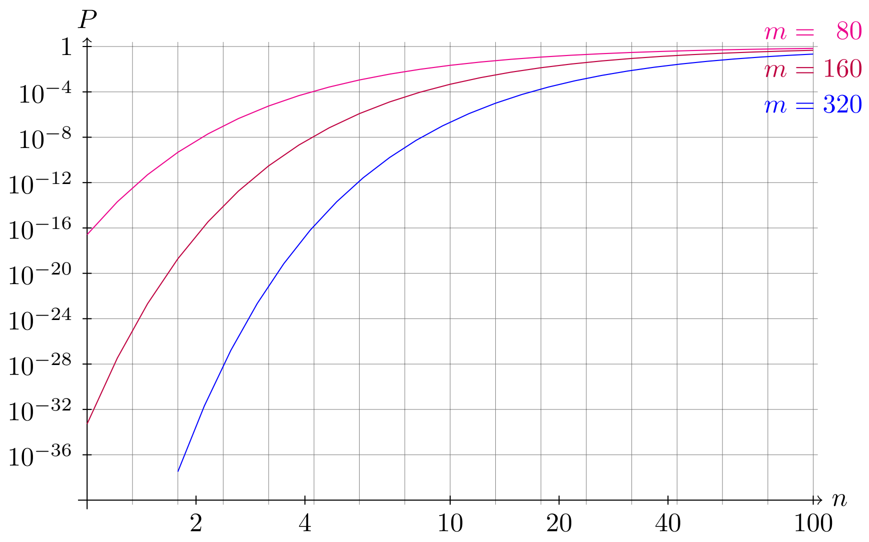


Figure 13: Approximative probability P of false-positives in Bloom filter

Algorithm 4.3.1.1 CMPT aggregating algorithm: $aggregate(G, G_1, \ldots, G_n, u, u_1, \ldots, u_n)$

```
**Require:** each CMPT ($G$, $G_1$, … or $G_n$) is a correct aggregated object. In particular each tree has at least $2L$ free bits.
**Require:** $u$, $u_1$, …, $u_n$ are vertices of the smallest generation of $G, G_1, \ldots, G_n$ respectively.
**Ensure:** return CMPT whose the center CMPT is $G$.
  $w_i \leftarrow H(u, u_i)$ for all $i$
  Let number leaves of the subtree of $G$ whose the root is $u$ (the order of the leaves when the tree is drawn): $1, \ldots, k'$.
  $j'_0 \leftarrow$ the lowest $j'$ such that the bloom filter of the tag $j'$ is empty. $j'_0$ is the **leftmost tag** with an empty bloom filter.
  **for** $i = 1$ to $n$ **do**
    Let number leaves of the subtree of $G_i$ whose the root is $u_i$ (the order of the leaves when the tree is drawn): $1, \ldots, k$.
    $j_0 \leftarrow$ the lowest $j$ such that the digest of the tag $j$ is free. $j_0$ is the leftmost tag with a free digest.
    Write $w_i$ in the digest of the tag $j_0$.
    Insert $w_i$ in the Bloom filter of the tag $j'_0$.
  **end for**
  **return** the resulting aggregated object.
```

The figure 14 is an example of CMPT aggregated object: the blue arrows represent the digest whereas the dotted magenta arrows represent the Bloom filter content.

Verifying algorithm (binary CMPT case) The second generic verifying algorithm (see algorithm 2.3.2.1) can be used with the following modifications: in the set E of the algorithms, the element are no more (T, h) (where T was a tree and h the potential parent id of the tree) but (T, h, B, b) where:

- T is a subtree (whose leaves are — read by interrogator — tags)
- h is the digest of the leftmost tag whose the digest was not used in the construction of G.
- B is the bloom filter of the leftmost tag whose the bloom filter was not used in the construction of G.
- b is a bit which is 1 by default.

And instead of searching for elements of E whose the second elements h are equal, the algorithm searches for elements (T, h, B, b), (T_1, h_1, B_1, b_1) such that $b_1 = 1$ and $h_1 \in B$. The algorithms hopes h_1

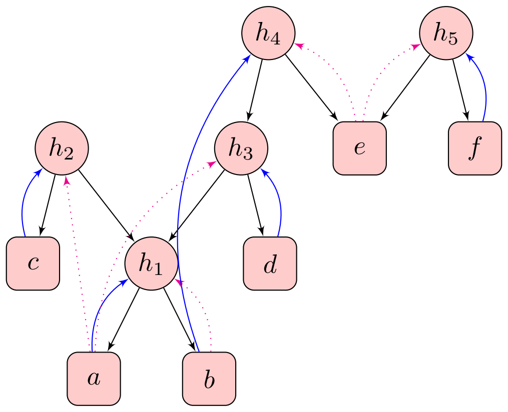


Figure 14: CMPT aggregating format example

is the parent of T and T_1. But it is not necessarily the case because h is not necessarily the parent of T. So the verification that h is a correct digest is very important. If the verification fails, contrary to the generic algorithm, h_1 and B must not be replaced by *null* but the algorithm shall replace b_1 by 0.

Notice also that if all maximum subtrees shall be found, the algorithm shall not removing (T, h, B, b) nor (T, h_1, B_1, b_1) from E if the digest h is correct but shall replace b_1 by 0.

Obviously, some optimizations can be made to improve performances of the algorithm.

General MPT cases When the MPTs are not binary, in the aggregating algorithm, w_i is stored in the leftmost free digest of each subtree whose the root is one of its child except the subtree included in G.

The verifying algorithm groups all quadruplets (T, h, B, b) with the same h. Normally the bit b is the same for all these quadruplets (this is an invariant easy to prove). The bloom filter of the group shall be considered empty (a group of quadruplets cannot be a centered subtree; considering the group has an empty Bloom filter is correct and enables to reduces the complexity of the algorithm). There is no other modifications to the verifying algorithm.

4.3.2 Analysis

Bloom filter parameters The Bloom filter parameters depend only on the number of insertions.

In our case, the maximum number of insertions is the maximum number of parents of a node.

The figure 13 shows that if each node has less than 4 parents and if the Bloom filter has 160 bits, the false-positive rate is about 10^{-8} which is enough sufficient for the verifying algorithm in most cases.

Security The proposition 3.3.1 shows that tree aggregated objects have really strong security properties, if the used hash function is second-preimage resistant.

The basic idea is that modifying an read-only aggregated object is complex because it indirectly lead to compute a second preimage.

Unfortunately, because of the Bloom filter, it is no more the case with CMPT format. Indeed, it may be sufficient to find a digest which is inside a Bloom filter (this is for example the case for adding the tag e in the figure 15). And such a digest can be found with only 10^7 tries (in mean, in some cases, with the parameters of the previous subsection).

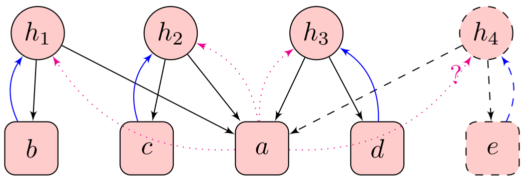


Figure 15: Adding a tag in a read-only CMPT aggregated object may be too easy. In this example, to add the tag e, it is sufficient that $h_4 \in B$ where B is the Bloom filter of the tag a.

Part II
Dependability

This second part approximately corresponds to the second part of my internship: the goal was to analyze dependability impairments and to try to find solutions to provide safe, secure and available aggregate based systems.

After describing main dependability impairments in the first section, theoretical cryptographic solutions and practical implementations are detailed. These two last sections are mainly applications of existing ideas to aggregates based systems. But I have also introduced some ideas I think novel[8], in particular: use of IBE to provide unclonability feature (section 6.2.3 — but this idea has maybe strictly no interest), aggregated objects authentication (section 6.4.2) and use of MAC instead of hash functions for aggregates (section 6.4.3).

Three other important subparts of my work are in appendices: a (non exhaustive) list of current RFID tags (annex F), a short summary about short signatures using pairing based cryptography (annex D — this summary focuses on practical choice of implementation and elliptic curves) — and benchmarks of cryptographic primitives (annex E).

5 Dependability impairments

This section mainly focuses on attacks against aggregates based systems (like UbiPark — section 1.3.2 — or UbiQuitus — section 1.3.2) and their descriptions with concepts and terminologies defined in the book [31]. More precisely, failures, errors and faults, (of a whole aggregates based system) related to attacks are specified. Some other faults are also shortly described to show that dependability impairments are not limited to attacks.

These dependability impairments also depend on the considered application. This section will mainly use UbiPark and UbiQuitus examples. However a lot of failures and faults are common to almost all aggregates based systems.

5.1 Failures

Aggregates based systems have been created to provide safety and sometimes also security. In particular they enable to prevent theft (for UbiPark) or real life attacks (for UbiQuitus); and they can also enable to control access to a service (in UbiPark case).

Thus main failures are: substitution, theft, real life attack and unauthorized use of a service. But there are not the only failures. Deny of access to authorized person and privacy leaks are also important failures.

5.1.1 Substitution, theft and direct real life attack

As explained above, these failures are the reason why UbiPark and UbiQuitus exist.

In UbiPark, an attacker should not be able to exchange or steal a bike, or a part of a bike. In UbiQuitus, an attacker should not be able to take a bottle of nitroglycerin (or to exchange it with a bottle of water) and to use it later to commit a bomb attack (in the safety area).

The main dependability attributes affected by this failure are safety and security.

5.1.2 Unauthorized use of a service

This failure occurs when the verifying system give an access to an unauthorized person. For example, in UbiPark, when the verifying system let enter a user who have not subscribed UbiPark service.

This failure may lead in more critical failures (such as previous failures) if other faults occur: an attacker in the shed can more easily do attacks than an attacker outside the shed.

The main dependability attribute affected by this failure is security.

[8]I cannot be sure an idea is original. For instance, I discovered idea described in section 6.2.2 before seeing that researchers have already found it.

5.1.3 Deny of access to authorized persons

This failure occurs when the verifying system denies access to an authorized person.

The main dependability attribute affected by this failure is availability.

This failure often has no catastrophic consequences, contrary to the two previous failures.

5.1.4 Privacy leaks

Privacy leaks occurs when an attacker can retrieve information about users of the aggregates based system. Even if such a system does not need any external database and, in this meaning, anonymity is enforced[9], an attacker may link a tag id with a person. For instance, the attacker may want to watch the moves (by bike) of a known person (his wife or her husband for example): in this case, he just need to store all tag ids of this person's bike and put some tag interrogator in the city.

This failure will not be treated in this report but is really complex considering the previous example and according to [4].

The main dependability attribute affected by this failure is security.

5.2 Errors

Actually errors are quite only a inconsistency between the reality (the real aggregated objects created by an genuine and authorized systems) and the state (the set of aggregated objects) a verifying algorithm has detected.

But there are a lot of kinds of inconsistencies.

Some can be detected. For example, in UbiPark, if the shed is monitored (tags are regularly read and aggregates are verified), the destruction of a tag produces an inconsistency in the set of aggregated objects, because one of the aggregated object of the shed (a bike) is no more integral. In this case, the verifying system can detect the destruction of the tag and can throw an exception. For example, the keeper is warned and can program a new tag.

But some errors cannot be detected. For instance, in UbiPark, if an attacker exchange tags of two bikes, the verifying system cannot detect it. The created error remains latent until the attacker exit the shed with the exchanged bike and produces a failure: a substitution (which is a form of theft).

5.3 Faults

This subsection mainly focuses on intentional external human-made faults, another name for attacks. The second, less important, part gives a very short overview of mechanical, hardware and software faults.

5.3.1 Intentional external human-made faults

The described failures can often be produced (indirectly) by an intentional human-made fault.

Physical attacks These kinds of faults are the less subtle fault but also the easiest ones. An attacker does not need any knowledge in RFID to perform these faults.

Nowadays and in general public systems with aggregation, these faults can be considered as the most common faults.

There are three common main threats:

- an attacker can physically destroy a tag. It can lead to the following failure: access denied to authorized persons or more generally to any availability failures.
- an attacker can prevent a tag from receiving waves from a tag interrogator by putting it in a Faraday cage, for instance. The only difference with the previous attack is that this operation is easily reversible: the cage has just to be removed.

[9]An aggregates based system does not need to store identity information of owners in an on-line (connected to controllers) database nor in tags. An off-line database with a list of users of the service and their tag ids may be used. For instance, in an UbiPark application, it can enable to find a bike thief who uses a genuine tag to enter the shed (in this case, the controller shall log ids of all users who enter or exit the shed). However this database cannot be easily attacked, because it is off-line.

- an attacker can physically move a tag: unstick a tag and stick it on another fragment. It can lead to availability failures but also to substitutions failures.

If it is possible to buy aggregated tags not attached to an object (for example, if it is possible to buy UbiPark tags on the Internet to put on a bike), there are more possible attacks. For instance, in UbiPark, an attacker can destroy tags of the bike he wants to steal and put on it the bought tags.

Possible solutions using special features of tags to prevent this faults can be found in section 7.1.2. Monitoring (or regular checking of tags) can also partially solve the problem since it may be possible to detect when a tag is destroyed and to warn a guard (thanks to an alarm).

Obviously other physical faults can be committed against a specific application. For instance, in UbiPark, the attacker can simply break the door to steal a bike.

This example shows that it is often useful to add classical protection (like video surveillance or alarm system) to an aggregates based system.

RF attacks These attacks require some knowledge in RFID and specific hardware. But, since RFID will be more and more used, anyone may have a tag interrogator in their mobile phones (for example) in a few years time.

An attacker can:

- write random data in an already aggregated tag. It may lead to availability issues.
- write random data in an new tag (create a fake tag) to disturb the system. It may lead to availability issues.
- send high power waves to prevent a genuine tag interrogator to communicate with genuine tags. It may lead to availability problems. Furthermore this attack cannot be prevented except by using an external system (an alarm which calls a guard).
- write aggregating data in a tag "from scratch" (without cloning). It may lead to substitution, direct theft, direct real life attack or unauthorized use of a service. For instance, in UbiQuitus, an attacker can modify tags such that a bottle of nitroglycerin, previously coupled with him, is no more coupled with him. Thus it can put this bottle anywhere: he does not need anymore to be near this bottle.
- clone a tag. It may lead to substitution. Indeed, if tags are clonable, destroyable but not movable (i.e. which cannot be unstick and stick again on another object), it is nevertheless possible to exchange two tags: an attacker just need clone the two tags, destroy the two original tags and put the two copy on the fragments. Actually, this is maybe the easiest and most effective attack if tag are clonable.

Some theoretical solutions to prevent these attacks are given in section 6. The practical implementations are described in section 7. Actually this report focuses on this subject.

All proposed solutions use cryptographic protections. These protections shall be safe: it shall not be possible to break them, that means to produce a failure (at this level), which can then enable an attacker to do one of the previous attacks. To break cryptographic protections, an attacker can (first attacks are the more probable):

- read genuine tags: this is the easiest attack.
- perform attacks on a genuine tags such as side-channel attacks.
- passive eavesdrops a communication between a genuine tag and a genuine tag interrogator.
- perform attacks on a genuine tag interrogator such as side-channel attacks.
- use genuine tags to perform man-in-the-middle attacks (see section 6.2).
- use genuine tag interrogator to perform man-in-the-middle attacks (see section 6.3).

Actually, these attacks are common to all cryptographic tags.

5.3.2 Mechanical, hardware and software faults

Some mechanical, software or hardware faults may happen and may produce one of the previously described failures.

An aggregation application should be able to work (in a degraded mode) if one part of the application is broken down (for instance, the tag interrogator or the door of an UbiPark shed). Furthermore, when the issue is fixed, the system shall work again.

This description is very generic and error processing[10] depends a lot on the application. The end of this subsection is devoted to UbiPark.

In UbiPark, there may be the following mechanical faults: broken door, broken controller computer, broken tag interrogator, . . . The two first faults are described below.

If the door is broken or blocked, anyway can enter or exit the shed. Firstly an alarm should be set off and a guard should be warned. But it is not sufficient.

The content of the shed can have changed: some new bikes may be in the shed and some old bikes may have disappear. If there is only a tag interrogator near the door to control entrance and exit, persons who have put their bikes when the door was opened cannot get back their bikes and persons who have get back their bikes during this period can enter the shed without any bike. Therefore, a tag interrogator in the shed shall monitor the bikes.

If the controller computer is broken, an alarm should be set off. If security is more important than availability, the door shall be closed and otherwise, the door shall be opened.

[10]The errors created by the mechanical and hardware faults shall be recovered or at least partially compensated.

6 Solutions

This section focuses on the theoretical cryptographic solutions to prevent RF attacks (section 5.3.1).

Implementations of these solutions are treated in the section 7.

6.1 Keys and cryptosystems

6.1.1 Symmetric and Asymmetric cryptosystems

There are two main kind of cryptography: symmetric cryptography and asymmetric cryptography.

With a symmetric cryptosystem, a key is shared by all users[11]. For encryption cryptosystem, this key is used for encryption and decryption. For dynamic authentication cryptosystem[12], the same key is used by the user who wants to prove its identity and by the user who wants to verify this identity. For message authentication code (MAC) algorithm[13], the same key is used to create the MAC and to verify the MAC. So there are the following issues:

- it is impossible to distinguish users of a symmetric authentication system (for example, a MAC can be issued by any person who has the key).
- verifying a MAC (or play the role of the verifier in a dynamic authentication) requires the shared key.
- if one person uses badly its key (for example issues MAC of bad messages), the key has to be changed and all previous encrypted or authenticated data must no more be used.

Therefore it is very important to ensure high protection of chips and computers which have the shared key in their memory. If an attacker can theft such a computer and perform successful physical attacks, the security of the whole system collapses. A solution can consist in often changing keys. But in most cases, if it is possible to change keys in controllers, rewriting all tags is quite impossible. A compromise solution can be to change keys each month and to use only the last five keys, for example. In this case, an attacker can illegally use the system only for 5 months (if the attacker has performed only one successful physical attack).

Asymmetric cryptography solves most of these problems. In return, asymmetric cryptography is often more complex and need more computing resources and more space to store data. With an asymmetric cryptosystem, each user generates a private key and a public key. The private key is kept secret whereas the public key is distributed to all other users. With the private key, the user can decrypt or sign messages or can dynamically prove to another user that he is the user related to a given public key. With the public key, any user can encrypt messages, verify signature or play the role of verifier in dynamic authentications. The private key is needed to decrypt and sign data and to play the role of prover in dynamic authentication.

With an asymmetric cryptosystem, if a user behaves badly, one can just revoke its public key (that means it is said this public key is related to a compromised private key). If a private key shall be shared by a lot of users (for example by all controllers), there are fewer advantages of asymmetric cryptosystem and symmetric cryptosystems may be used since they are faster and produce shorter data.

6.1.2 Digital certificate

A digital certificate enables to identify a person or a web site (for example). In particular, it contains :

- a private key (only for the version of the certificate's owner) generally for an asymmetric signature algorithm
- the corresponding public key
- the owner's description (for example domain name for Internet's certificates)

[11]Here an user is a tag, a controller or a tag interrogator.

[12]A dynamic authentication cryptosystem enables an user to prove to another user it knows a secret

[13]A MAC is a piece of information added to a data to authenticate the data. It is close to a signature in the real world but there is a big difference: anyone who can verify a MAC can also issue a MAC.

- the expiration date
- a possible signature by another user

Classical X.509 certificates are signed by a Certification Authority (CA) which ensures the validity of the certificate (the fact the owner of the certificate corresponds to the given description). Certificates can also be self-signed. It is the case for CA's certificates.

The CA can also revoke certificates: the certificates become no more valid (for example, when the owner has lost its private key). Revocation can be made by publishing lists of revoked certificates.

Certificates may be hierarchical: the CA sign several certificates for users which can sign other certificates, etc. So the system is very flexible. If an user behaves badly, the CA (or the user who signs its certificate) can revoke its certificate. The data signed by the bad user cannot be used anymore but there are no other problems.

The CA shall never be compromised, otherwise all certificates become unusable.

6.1.3 Key storage and shared key

Tag memory is often very limited. Storing a certificate (corresponding to a tag's signature for example) can be very complex. In most cases, the following solution can be used: all used public keys are stored in each controller and a small identifier is assigned to each public key. Tag memory can so store only this identifier.

But this solution is less flexible than certificate: in particular it forces each verifying system to have a database of all public keys. If each tag shall have a different public key (or private key for symmetric cryptosystem), the needed controller's memory may be very huge. In this case, certificates (necessarily with asymmetric cryptosystem) may be stored in the tag.

This idea is also useful when symmetric encryption is used for example: the identifier of the key used by encryption algorithm is saved (as a plain text). It makes easier changing shared key.

6.2 Unclonable tags and authentication

As seen before, cloning a tag is one of the most critical issue of an aggregates based system: it enables the attacker to substitute objects or to use an unauthorized service.

If tag contains only memory, cloning a tag is really easy: the attacker just needs to have an empty tag and to copy data from the original tag to the new tag. If manufacturers do not allow to write some memory bank (as it is the case with a lot of commercial tags), it is possible to simulate a tag with the appropriate hardware. In this case, tag can be a big electronic device (or electronic card) and may need a battery.

Password authentication of tag may seem to solve the problem since it prevents an attacker from reading tag: only authorized interrogators can do it. But this protection may be insufficient because the attacker can eavesdrops one or more communication between an authorized interrogator and a tag and get the password.

Actually even tag authentication with a more complex mechanism (for example zero-knowledge proof) is insufficient as soon as the secret (used by authentication) is shared by all tags. Indeed an attacker can use a tag simulator (see figure 16) with a genuine tag (not the tag he wants to clone but another tag he has legally bought for example) to simulate any tag (and in particular the tag the attacker wants to clone):

- if authentication is requested, the tag simulator uses the genuine tag to correctly answer
- if normal data read is performed, the tag simulator sends data of the tags to be cloned

This kind of attacks is called a **man-in-the-middle attack**.

Hence we propose several solutions:

- Randomized encrypt data from tag to reader thanks to a random data provided by the reader.
- The tag contains a secret key directly link to its id and prove it knows it to the reader without revealing it (with a zero-knowledge proof for example).

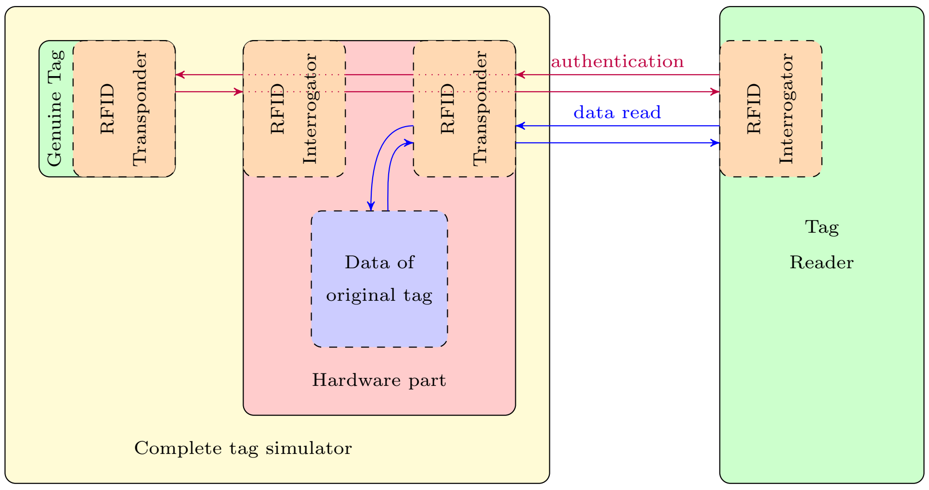


Figure 16: Tag simulator for cloning any tag with only authentication

- The tag contains a secret key directly link to its data thanks to an identity-based cryptography scheme and prove it knows it to the reader without revealing it (with a zero-knowledge proof for example).
- The tag uses a Physical Unclonable Function (PUF).

With the second method, tag is not really unclonable, just a part of the tag is unclonable: the id. But it is often sufficient (see remark 6.4.1). We will say tag has an unclonable id or unique id.

Remark 6.2.1. *Simpler (and cheaper) methods can be used if the security level is not very high: password authentication (section 6.6) for example.*

This section only contains advanced solutions for a very high security level.

6.2.1 Randomized encryption

Data from tag to reader are ciphered using a random number chosen by the reader. Hence each time a reader requests tag data, transmitted data are necessarily different, if reader choose a real random number (not always the same). Random number must not be chosen by tag because a tag simulator could choose always the same (the number chosen by the original tag when the attacker eavesdrops).

Here are three examples of protocols. These protocols may not be really secured and shall not be used as is (except TLS and SSH which are widely used schemes) but give the basic ideas.

For each protocol, the private keys are stored in the tag such that only the tag microprocessor can read the keys: an interrogator (and a controller) cannot access this memory area. Section 6.2.5 describes a very secured way to do this.

With symmetric encryption All tags and readers share a private key K of a symmetric cipher algorithm. e_K is the encryption function and d_K is the decryption function.

Here is an example of protocol to read data from tags:

1. The reader sends random number n.
2. The tag sends $a = e_K(n|data)$ where $|$ is concatenation and $data$ are the data of the tag.
3. The reader computes $d_K(a) = n'|data$ and verify that $n' = n$.

With asymmetric encryption All tags share a private key[14] sk for asymmetric encryption. All readers know the corresponding public key pk. e'_{pk} is the encryption algorithm (which needs the public key) and d'_{sk} is the decryption algorithm (which needs the private key).

Suppose c_K and d_K are symmetric encryption and decryption function with a key K.

1. The reader chooses a random symmetric key K and send $a = e'_{pk}(K)$ to tag.
2. The tag computes $K = d'_{pk}(a)$.
3. Communication are ciphered by c_K and d_K. For example, the tag sends $b = c_K(data)$ and the reader computes $data = d_K(b)$.

TLS ([20]) is a widely used protocol based on this idea: the tag is the server whereas the tag interrogator is the client.

With signature and key-exchange All tags share a private key sk for signature. All readers know the corresponding public key pk.

A Diffie-Hellman key exchange (see Wikipedia article [47]) is made between tag and reader and tag signs its sent data (during the key exchange) as in RFC 4253 [41] (SSH protocol version 2) for example (the tag is the server and the tag interrogator is the client). The resulting key can be used to encrypt further communication as with previous solution.

Remarks The first algorithm uses a shared key between all readers and tags. So any reader can create genuine tag. This is potentially dangerous if an attacker has access to a reader. The two other schemes do not have this problem.

The described second algorithm does not provide tag authentication: a non-genuine tag can send random data (data will not represent an aggregated object but reader will consider the tag genuine). But TLS (which is a bit more complex) provides tag authentication.

TLS and SSH version 2 schemes are widely used and may be used as is (even if they are not proved secure). The first algorithm shall not be used as is: it may exist a lot of attacks.

6.2.2 Unique id with zero-knowledge proof

In [6, 13, 43], there is a solution which does not need a shared by all tags secret key. For each tag a private key and a public key for a zero-knowledge proof[15]of identity (or just a signing algorithm like DSA) is generated. The public key is the id of the tag whereas the private key is stored in the tag (such that only the tag microprocessor can read the key — more details can be found in section 6.2.5).

The tag can prove its id is authentic by proving it knows the corresponding private key. This proof shall not reveal the private key.

This solution has many advantages over the previous one:

- there is no shared key common to all tags
- the protocol between tag and interrogator can be a standard protocol (like C1G2 standard — see section 7.1.4) with an additional command which proves the authenticity.
- authenticity verification can be performed only when high level of security is needed. Indeed it may be possible some usages of aggregated object require a low level of security whereas other usage require a high level of security.

However there are also some disadvantages:

- id cannot be chosen (otherwise there is not protection !).

[14]A mechanism with certificates can also be used with restrictions seen in 6.1.3

[15]There are two kinds of zero-knowledge proofs: honest verifier zero-knowledge proof (like Schnorr one [42]) and general zero-knowledge proof (like Okamoto one [39]). With the first kind, an attacker who eavesdrops communication between a genuine tag and a genuine tag interrogator cannot learn any information about the private key (except information he can directly computes from the public key). With the second kind, an attacker who can make requests to the tag, cannot get any information about the private key. So general zero-knowledge proof shall be preferred when a high level of security is required.

- there is no authentication of the tag: any manufacturer can create such tags contrary to previous method.
- only id is protected.

The two last issues can be solved by adding a signature (or a Message Authentication Code) to the data (id included) of the tag (see section 6.4.1).

6.2.3 Unclonable data with identity-based cryptography

Identity-based cryptosystems are asymmetric cryptosystems where the public key can be any unique data (called identity ID). As explained in [49, 48], there is a trusted third party, called the Private Key Generator (PKG) which can generate private keys corresponding to identity ID. To operate, the PKG first publishes a master public key, and retains the corresponding master private key (referred to as master key). Given the master public key, any user can compute a real public key corresponding to any identity ID by combining the master public key with the identity value. To obtain the corresponding private key, the user authorized to use the identity ID contacts the PKG, which uses the master private key to generate the private key for the identity ID.

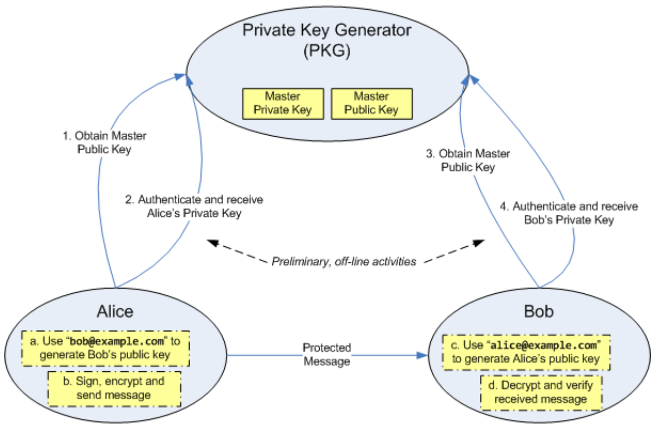

Private Key Generator (PKG)
Master Private Key
Master Public Key
1. Obtain Master Public Key
2. Authenticate and receive Alice's Private Key
3. Obtain Master Public Key
4. Authenticate and receive Bob's Private Key
Preliminary, off-line activities
Alice
a. Use "bob@example.com" to generate Bob's public key
b. Sign, encrypt and send message
Protected Message
Bob
c. Use "alice@example.com" to generate Alice's public key
d. Decrypt and verify received message

Figure 17: Illustration of identity-based cryptosystem (from Wikipedia)

In our case, the whole tag memory content is used as identity ID (and so indirectly as public key). Each time an aggregating system creates a new aggregated object, it asks the PKG for the corresponding private key and store this key in the tag (such that only the tag microprocessor can read the key).

The private key can then be used by the tag to dynamically prove its authenticity. In [56], there is an example of such a scheme. So the tag is really unclonable and authenticated.

Furthermore, there is no need of a tag certificate in the tag or a database on the controller contrary to classical authentication (see section 6.1).

Remark 6.2.2. *A variant with symmetric cryptography also exists and is easier: the key stored in the tag is a MAC (see section 6.1) of the data of the tag. The private key for the MAC shall be known by all controllers.*

The stored key can then be used by a symmetric authentication protocol. For example (this may be not secure and should not be used as is), the tag interrogator sends a random number and the tag authenticate this random number thanks to a MAC with its stored key.

In this mechanism, there are two MAC algorithms: one to create the private key which is stored in the tag and whose the key is shared by all controllers and another to prove the authenticity of the tag. The key of the second MAC is the key stored in the tag (the -first- MAC of the tag data).

6.2.4 Physical Unclonable Function (PUF)

According to [19], a Physical Unclonable Function (PUF) is a function (or more precisely an hardware circuit whose output depends approximatively only on the current input) that is:

- based on a physical system
- easy to evaluate (using the physical system)
- such that its output looks like a random function.
- unpredictable even for an attacker with physical access

A PUF often uses small differences between each integrated circuit (even with the same layout), for example path delays.

First part of [9] is an example of use of PUF f for authenticating each tag. A more general idea could be to save a lot of pairs $(c, f(c))$ (where c is a random entry of the PUF) in each tag interrogator (for all tags). Then a tag interrogator ask a tag to give the output of its PUF corresponding to some randomly chosen inputs c. Outputs of a PUF may depend a bit on external condition (like temperature). But this issue can be solved by accepting some error bits in the answer of the tag.

Unfortunately, this solution needs a lot of space on each tag interrogator to store all pairs $(c, f(c))$ for all tags; or each interrogator shall be connected to a big server which stores all these pairs.

But one advantage is that tags are really unclonable without the need of costly (and sometimes impossible to implement in small tags) cryptographic primitives. *Verayo* ([44]) buys this kind of tags.

6.2.5 Storing secret data, physical attacks and PUF

All previous solutions (except the last one) need to store a secret in each tag. This secret shall be readable only by the tag microprocessor.

If requested security level is very high, it is not recommended to use memory for this purpose because physical analysis of the tag's chip can enable an attacker to retrieve the secret. Then, if the secret is shared by all tags, the attacker can do what he wants.

Fortunately, PUF can provide a solution. Indeed opening a chip with a PUF will almost always change the PUF behavior. It is difficult to use directly a PUF because output of a PUF can depend a bit on external conditions. But there are ways to solve this problem. For example, in [43], the authors present a tag authentication with private key (signed by a certificate) stored in PUF. The interesting part for our purpose is the method to store the private key in a PUF: it uses an helper data and a special function which takes the helper data and the response of the PUF to compute the private key. The helper data normally leak very few bits and can be stored in a normal memory.

6.3 Memory write protection

As seen in 5, memory write shall be avoided in order to provide availability. Indeed if anybody can write anything in tags or can disable[16] a tag, an attacker can easily make the system unavailable.

A possible solution is to have **reader authentication** (not tag authentication as in the previous section). The basic solution is a password. In section 6.6, some advices on ways to use password are given. A better method (if high level of security is required) is to use a symmetric or asymmetric authentication scheme as those described in section 6.2.1.

However two points shall not be forgotten:

- Man-in-the-middle attacks (see section 6.2 — here a genuine tag interrogator can be used to do the authentication and then the attacker's tag interrogator can be used to write data in the tag) shall be (almost) impossible.
- Tags often cannot have a database of public keys of all authorized tag interrogators nor verify any certificate expiration's date (since most tags are passive and are power supplied only when a tag interrogator reads or writes them, they cannot maintain a clock). Hence reader's authentication is very complex. More informations can be found in the article [38].

Previous solutions need tag with some extra-features (a cryptographic microprocessor, a PUF or both). This is absolutely necessary to prevent cloning since memory only tag can be read and copied in another

[16]A lot of tags provide a "kill" feature which enables a tag interrogator to kill the tag, i.e. to permanently disable the tag.

tag. Thus one of the previous solutions (or equivalent solutions) shall be implemented if a high level of security is required. But it is not sufficient.

Now we will focus on supplementary solutions only using memory of tags: only the software of the controller will perform cryptographic operations.

6.4 Aggregating company authentication

6.4.1 Tag authentication

Tag authentication enables an aggregating company[17] to prevent unauthorized aggregating systems from creating aggregated object or fake tags (related to no aggregated object but seen as a part of an aggregated object by a controller). Hence an attacker cannot disturb a system with fake tags nor misuse the system (use of a service without subscription for instance).

The authentication can be dynamic or static.

With dynamic authentication, tag has a microprocessor with cryptographic primitives and can prove its authenticity to the reader. A zero-knowledge proof of identity (see footnote 15) or a proof with a signature of a random data chosen by controller may work. Some examples (more complex than necessary because they also provide unclonability feature) are given in sections 6.2.1 and 6.2.3.

Static authentication only uses tag memory: a small amount of data is added at the end of the data for aggregation[18] which proves aggregation has be done by an authorized aggregating system. If the used cryptosystem is symmetric, these extra-data are called a MAC (Message Authentication Code), otherwise it is called a signature (as explained in section 6.1).

On the one hand, with signature mechanism, it is possible to know which controllers has created each aggregated object and if a controller behaves badly, its public key can be revoked (see section 6.1). On the other hand MAC algorithms are generally a lot faster and produce a lot shorter message authentication (regarding memory space used in the tag) than signature cryptosystems. In addition, MAC algorithms often use either cryptographic hash functions or symmetric block cipher and these cryptographic primitives are used by other part of the controller's software (hash functions are often used in the aggregating and verifying algorithms).

Remark 6.4.1. *If tags have an unique id (see section 6.2), signature or MAC makes tag indirectly totally unclonable, as more complex solutions like identity based encryption scheme (section 6.2.3).*

Authentication and digest size

With tag authentication, the security properties seen in section 3.3 are indirectly enforced without need of a second preimage resistance of the hash function. An attacker cannot indeed create a tag with a new id (because the id is signed) nor change a tag digest. So using a tag or aggregated object authentication enables to reduce the size of digest (without reducing the security level) and remove the need of a write lock if availability is not important (because an attacker cannot change the content of a tag; otherwise the signature is no more available).

More precisely with authentication, the digest size is only determined by the required probability of collisions (see section 3.3).

However, if this trick is used, the signature (or MAC) shall always be verified whereas, if a second preimage resistant hash function is used, the signature can be verified only when high security level is required[19].

6.4.2 Aggregated object authentication

Instead of authenticating each tag, it is also possible to authenticate only complete aggregated objects. On the one hand, it uses less tag memory to store signature because the signature can be spread over several tags. On the other hand, an attacker can create fake tags and disturbs the system because individual tag

[17]An aggregating company is any company which bought aggregates based system. Nowadays, there is only one aggregating company: SenseYou (`http://www.senseyou.fr`)

[18]Data for aggregation are data used by aggregating and verifying algorithms: id, digests and relative depth with second tree format. Actually, it is possible to put other data in tag, for example a timestamp. These data can also be signed.

[19]There may have several uses of the same aggregated object which may require different security levels.

are not signed. Furthermore, aggregated object authentication is based on the security of aggregating object (see section 3.3), in particular a second preimage resistant hash function is required.

We will focus on signing tree aggregated object (CMPT will not be discussed here).

A tree (or subtree) is said **signed** if it is possible to get (from memory of one or more tags of the subtree) a signature of the id of the root of the subtree or a signature of a known plain text which contains (at a known place) the id of the root of the subtree. The algorithm will always sign concatenation of all tag data (including id but excluding the future signature part — concatenation may lead to critical flaws, for better methods see section 2.2.2).

A tag is said **indirectly signed in a subtree** if tag data are a part of a known signed plain text and if the corresponding signature can be retrieved from the tags of the subtree.

The goal is that each (not reduced to a tag) subtree is signed.

A first idea may be to use the same methods as for first tree format (section B.2): for each subtree, the signature of all tags of the subtree is computed and a part is stored in each tag depending on the depth of the tag in the subtree. A possible solution is to choose that the part size stored in a tag of depth d is $\frac{1}{2^d}$. It perfectly works with the same depth limitations of the first tree format[20]. But it does not really improves the space used by signature since with trees with high height, the space needed to store all signatures is nearly the space needed to store one complete signature. But if each node of the tree has a degree greater or equal than k with $k \geq 3$, the part size stored in a tag of depth d can be $\frac{1}{k^d}$. In this case, it effectively reduces the memory usage.

There is another method which always divides by two the space needed for the signature but which works with second and third tree format (but maybe not the other formats). Optimizations with non-binary trees may again improve space usage.

More precisely, each tag stores the half of a signature and for all aggregated object (or tag) the following invariant (for the third tree format) is verified: each (not reduced to a tag) subtree is signed, the leftmost tags with a free second digest is either indirectly signed in the subtree or can store an half of a signature.

For the third second format, the invariant is: each (not reduced to a tag) subtree is signed, the leftmost tags are either indirectly signed in the subtree or can store an half of a signature, and the rightmost tags verify the same property.

The algorithm (for the second tree format with left/right distinction — it is easy to adapt to other versions of second tree format and to third tree format) is described in algorithm 6.4.2.1 and works recursively exactly like aggregating algorithms. In order to make easier the algorithm's description, tree are supposed binary.

Example 6.4.2. *The figure 18 gives an example of signed aggregated object. The snake blue lines denote signature: if two tags are linked with a snake blue line, each tag contains half of the signature of the concatenation of data of these two tags.*

The easiest way (but maybe not the fastest) to verify a signed aggregated object is first to apply the verifying algorithm of section 2.3. Then simulate the algorithm 6.4.2.1 to find where the signature of the root digest *of each subtree* is stored and verify it.

Remark 6.4.3. *The remark 3.3.3 enables to store signed data thanks to aggregation, even if signature of the data is too long for a tag memory (but can be stored in two tags).*

6.4.3 Using MAC algorithm instead of hash function

There are another complementary way to use MAC: the hash function (used by aggregating or verifying algorithms) can be replaced by a MAC algorithms. In this case, the private key is shared by all controllers.

There are two main advantages. Firstly, only a genuine controller can create aggregated object[21].

Secondly, adding or replacing a tag in a read-only aggregated object becomes a lot more difficult. Indeed, with a perfectly safe hash function (it may not exist but let suppose currently used hash functions have this intuitive property) with n bits output, finding a second preimage needs to try about 2^n different

[20]See annex B.2. Advanced version of first tree aggregating format can be used. This enable to remove the depth limitation

[21]This property is obtained by almost all solutions of the section 6. However using a MAC algorithm instead of a hash function is maybe the cheapest (requiring the lowest resources) way to do it.

Algorithm 6.4.2.1 Aggregating algorithm for the third tree format with signature: signedAggregate(T_1, T_2)

```
Require: $T_1$ and $T_2$ are the trees to be aggregated
Require: $T_1$ and $T_2$ are correct signed aggregated object, in particular the left digest of the leftmost tags
    and the right digest of the rightmost tags are not used and the signature invariant (see section 6.4.2) is
    verified.
Require: Id of $T_1$ is less or equal to id of $T_2$.
Ensure: return signed aggregated object whose children of the root are elements of $T_1, T_2$ and which
    verify the signature invariant.
  $id_1 \leftarrow$ id of $T_1$
  $id_2 \leftarrow$ id of $T_2$
  $h \leftarrow$ the digest $H(id_1, id_2)$ (see section 2.1)
  Write the digest $h$ in the leftmost free digest (tag $u_1$) of $T_1$ and update the potential signature which
  uses $u_1$ data (a part of this signature is stored in another tag — another structure may be stored in
  the controller's memory to know, to what plain text data, each tag's signature part refers).
  Write the digest $h$ in the leftmost free digest (tag $u_2$) of $T_2$ and update the potential signature which
  uses $u_2$ data.
  if $u_1$ is not indirectly signed in the subtree $T_1$ and $u_2$ is not indirectly signed in the subtree $T_2$ then
    Compute the signature $sig$ of the concatenation of all data of $u_1$ and $u_2$.
    Store the left half of $sig$ in $u_1$.
    Store the right half of $sig$ in $u_2$.
  end if
  return the aggregated object which corresponds to $L$
```

inputs. If n is big enough, this computation is very complex but can be performed on any computer without any access to a verifying system. But, if a perfectly secure MAC algorithm with n bits output is used instead of an hash function and if the key cannot be recovered, trying 2^n different inputs require to do 2^n (or 2^{n-1} in mean) requests[22] to a genuine verifying system. So if a verifying system does not accept more than 1 request per second (for instance), a brute force attack against an aggregated object which uses a MAC algorithm needs at least about 2^n seconds whereas such an attack against an aggregated object which uses an hash function requires only 2^n computations of the hash function (and each computation may take only a few milliseconds — furthermore these computations may be distributed on a huge number of computers).

Using a MAC enables to reduce the size of the node id (without reducing the security level).

Remark 6.4.4. *It is also possible to use a signature (of the digest) instead of the digest (or the MAC) itself. Each aggregating system has a different private key and all verifying systems have all corresponding public keys. Even if this solution is more flexible (than MAC) for key management, it has two disadvantages. Signature size is often much bigger than MAC size and signature verification is slower than MAC computation. Unfortunately aggregating systems often need to store two digests (or signature in this case) and verifying system may have to compute many digests in some cases.*

Remark 6.4.5. *Using MAC instead of hash function does not prevent an attacker to create fake tag.*

6.5 Encryption

Encryption of the tag data avoid unauthorized readers to parse data of tags and brings so the following advantages:

- only authorized readers can create aggregated objects.
- an unauthorized reader cannot say if objects are aggregated or not (privacy feature).

[22]Request is a very generic term but it cannot be easily specified. Actually, the issue is to prevent the attacker from verifying more than one MAC in a given unit of time. MAC verification cannot be normally directly performed but they often can easily be indirectly performed (for example by putting tags with special data near the tag interrogator and by watching its reactions).

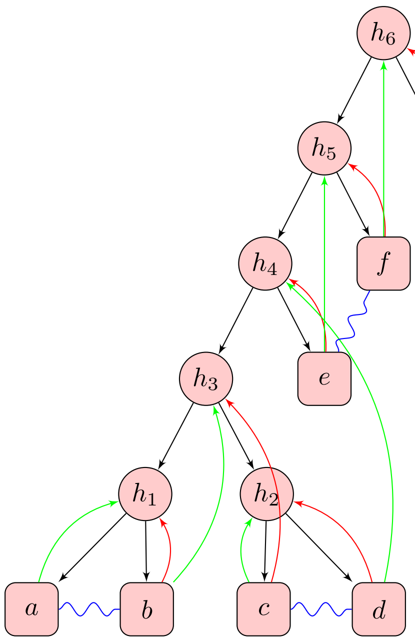


Figure 18: Signed aggregated object (in second tree format with left/right distinction

- a company can prevent other companies to sell controllers compatible to their products[23].

The first point can be performed by a company authentication (see 6.4). But symmetric encryption is often a lot faster than signature (but not than MAC).

Warning 6.5.1. *Generally encryption does not provide authentication. An attacker can make a fake tag with random data (instead of encrypted data) and he can so disturb the system (the tag is seen as a part of a aggregated object by the controller although it is just a fake tag).*

Asymmetric or symmetric encryption algorithms can be used. However it does not seem very useful to use asymmetric algorithm because the private key (used for decryption) shall be shared by all RFID readers anyway and asymmetric encryption algorithms are often slower than symmetric ones (i.e need more computing resources) and cipher text are often longer than plain text (for example, for El Gamal encryption algorithm, cipher text size is twice plain text size).

If signature (see section 6.4) is also required, signcryption can be a good alternative to symmetric encryption and asymmetric encryption. Signcryption is a cryptographic primitive which simultaneously sign and encrypt (in an asymmetric way) a plain text.

But separated symmetric encryption and signature have the following advantage: a cheap controller can only decrypt the tag without verifying signature whereas a state of the art controller can decrypt tag data and verify signature.

If MAC (see section 6.4) is also required, authenticated encryption can be used. Authenticated encryption is a cryptographic primitive which simultaneously performs a MAC and encrypts a plain text. There is often only one private key for these two operations. Authenticated encryption is something like a symmetric signcryption.

When neither signcryption nor authenticated encryption is chosen, there is another choice to do: whether the tag is first signed (or authenticated by a MAC) then encrypted or if the tag is first encrypted and then signed (or authenticated by a MAC — signature or MAC is not encrypted). The second solution brings two advantages: it needs to encrypt a smaller amount of data and signature can be verified without decrypting data. The first solution hides the signature which may be useful. In particular, it prevents an attacker who knows the signature public keys (used by each controller) but not the encryption private key from saying which controller has created the aggregated object.

[23]It is not only a question of monopole on the technology. Regarding the security, it can be very important. For instance, the other company may not verify signatures or MACs, or the public keys database may be not enough often updated.

6.6 Use of password

Passwords are the simplest way to do an authentication. But, as explained in section 6.2, passwords can be eavesdrop. If an attacker manages to get a password, he can do the same things as a genuine reader.

So some rules can be recommended:

- reduce the number of times a password is sent over the air.
- password memory (write or read) lock should be used only when permanent lock cannot be used (when a tag shall be used multiple times).
- passwords should not be the same for all tags.

In order not to use the same password for each tag, there are (at least) two possibilities:

- store the password in a secured tag (with real authentication and encryption). Most aggregates based applications have this kind of tag. For example, in UbiQuitus (section 1.3.1), it is recommended that the persons have a secure badge.
- the password of each tag is a MAC of the data. The key of this MAC shall be different from the keys of the potential other MACs of the aggregates based application.

The second possibility enables to do a really simple authentication of the tag and, if eavesdropping is impossible, it prevents from cloning tags. Indeed an attacker does not have access to the password and so cannot copy the tag.

If password are used to prevent memory write to unauthorized users, the second possibility can be used and the password (and so the write operations) are only made in areas where there is no eavesdropper.

7 State of the art

The section 6 provided only theoretical solutions without any detailed implementation (except for tag authentication and unclonability). This section focuses on current tags and on practical choices of algorithms used by controllers. The last subsection gives two possible practical implementations.

7.1 Tag features

Here is presented a non exhaustive list of current RFID tags.

7.1.1 Tag class

There are a lot of standards for tags. EPCGlobal^{TM} is an organization which tries to do a common standard. Notations and definitions of EPCGlobal^{TM} are used in this report.

According to EPCGlobal^{TM} in the whitepaper [25], there are four classes of tags:

- class 1: identity tags. They are passive tags, that means, they are powered by waves of the tag interrogator.
- class 2: higher-functionality tags. These tags are also passive but provide higher functionalities such as authentication.
- class 3: battery-assisted passive tags or semi-passive tags. These tags also have a power source that may supply power to the tag and/or to its sensors. But they still communicate passively: a class 3 tag requires an interrogator to initiate communications.
- class 4: active tags. These tags have access to a transmitter and a power source and may initiate a communication without an interrogator. The read distance may be a lot bigger than for other class tags.

The report focuses on passive tags (class 1 and class 2) because there are most common tags and are cheaper than other tags.

However sometimes active tags may be very useful because their read distances are bigger. For example, in the UbiPark application, it may be possible to add an active tag to each bike. This active tag sends its id each minute (for example). It enables to easily watch bikes in the shed. Using only passive tags may be possible but since read distance are shorter, it may require using a lot of tag readers inside the shed.

The active tags does not necessarily need an user memory. But its id shall be included in the id (digest) of the node corresponding to the bike. By using the passive tags, it is possible to verify the integrity of the bike at the entrance and the exit of the shed (including the id of the active tag). By using the active tag, it is possible to verify the bike is not stolen inside the shed. However, it does not prevent an attacker from destructing passive tags of the bike nor from stealing a part of a bike (like a wheel), contrary to a monitoring of all passive tags inside the shed.

7.1.2 Physical features

With aggregates based applications, RFID tags are attached to a physical object or to a human being. For example, in UbiPark (see section 1.3.2), several tags shall be sticked on a part of the bike: wheel, saddle, ... Another tag shall be carried by the owner of the bike.

The second kind of tags cannot be easily stolen, or more precisely the risk of theft is the same as the risk of theft of identity card or credit card.

But the first kind of tags may be easily stolen, unsticked and sticked on another bike or simply destroyed if there is no protection. So an attacker can make failures in the whole system. More precisions can be found in section 5.

Therefore tags should be neither destroyable nor being unsticked. It seems very difficult to provide these two features, at least if the attacker has a lot of time and can use, for example, acetone to remove the glue or a drill to destroy the tag.

The first physical feature may be provided by rugged tags and very good glue. However it may become possible to unstick a tag and to put it on another object. If availability is the first criteria, this solution may be used: but it must be kept in mind that there are high risks of theft of objects: for example, with UbiPark application, bike can be stolen if tags can be unsticked.

The second physical feature (preventing from unsticking tags) is provided by some tags, called destructible tags: when an attacker tries to remove such a tag, the tag is destroyed (see [27]). Unfortunately, it makes tags very destroyable. Availability is lost but this feature may be necessary to guaranty safety and security.

In summary, most applications need destructible tags but in some special cases, when availability is very important or when an attacker cannot unstick tags for other reasons (for example if he cannot have enough time to unstick tags but if he could have enough tag to use a hammer to destroy all tags) rugged tags and very good glue may be preferred.

7.1.3 Frequencies

According to [55], there are four frequencies ranges for RFID tags:

Frequencies	Europa and Africa	America	Asia and Oceania
BF	125 kHz	125 kHz	125 kHz
HF	13.56 MHz	13.56 MHz	13.56 MHz
UHF	865-868 MHz	902-928 MHz	902-928MHz[24]
SHF	2.446-2.454 GHz	2.427-2.47 GHz	2.4-2.4835 GHz

But currently most tags are HF or UHF.

The fact that UHF frequencies are not the same in all geographic areas is not really an issue. Indeed, UHF tags often accept a wide range of frequencies and they use the frequency provided by the tag interrogator. Only tag interrogator shall be changed when an aggregates based application is deployed in different geographic areas.

HF tags are not completely standardized. There are three main standards: ISO/IEC 14443, ISO/IEC 15693 and ISO/IEC 18000-3. But HF tags of different manufacturers are often incompatible because security mechanisms are different. Furthermore read distance is not huge: often less than 1 meter. However, a big advantage of HF tags is that they often support advanced security protections contrary to UHF tags.

Almost all current UHF tags are EPCGlobal^{TM} class 1 generation 2 tags (called C1G2 tags). The standard is available on the Internet: [26]. Some manufacturers add some features to their tags but tags can still be read with a tag interrogator which does not support these features.

According to [2]:

> Early UHF tags occasionally encountered problems around materials like metal and liquids. This led HF proponents to assert their technology was more reliable in those environments. Due to technology advances made in the last few years however, Gen 2 UHF tags now perform as well as or better than HF around these materials, with the added benefits of universality and costing less on a per-tag basis.

So in a lot of applications, UHF C1G2 tags should be used for tags sticked on a physical object and HF (more secured) tags should be used for human beings.

7.1.4 Software and hardware features

This section is a summary of the appendix F which contains a (non-exhaustive but relatively representative of the current state of the art) list of HF and UHF RFID tags.

HF tags There are the three main standards:

- ISO/IEC 14443 (A or B): read distance is often about 10cm
- ISO/IEC 18000: read distance is often about 1.5m
- ISO/IEC 15693: read distance is often about 1.5m

[24]Depends a lot on the country. For example: Japan 952-954 MHz, Oceania 910-914 MHz

Generally, only ISO/IEC 14443 tags provide advanced security features (more complex than passwords — section 6.2 gives a lot of examples of such security features) except for Legic Advant tags which are ISO/IEC 15693 tags but have some cryptographic primitives (but their read distance is about 70cm).

All ISO/IEC 14443 tags are not equivalent. Some tags are a lot more secure than other. It is recommended to use tags with published (and known secured) cryptographic algorithms. Proprietary cryptographic algorithm often have flaws (but are used because they are cheaper than well-known primitives). For example the MiFare Classic cards do not provide any security (see article [17]).

Known symmetric cryptographic primitives are DES, 3DES and AES. DES shall not be used when a high security level is required, since it is possible to break it in less than a single day thanks to 16 RIVYERA machines ([54]). For 80 bits security (see section 7.2 page 39 for a definition of bits of security), 3DES or AES may be used.

Main known asymmetric cryptographic primitives are RSA, DSA, Schnorr signature, zero-knowledge proofs like Schnorr and Okamoto, and elliptic curve variants of these algorithms (except RSA).

Furthermore a Random Number Generator (RNG) is often required for authentication protocols. The quality of the RNG is very important and pseudo RNG shall not be used in most cases. True RNG are preferable.

Recommended security strength and corresponding key length can be found on this website: [53] and in this NIST document: [37].

However even if good primitives with good keys are used, there may be flaws due to protocols (some protocols are insecure even if they use secure primitives) and side-channel attacks (see [51]). Since manufacturers do not give enough informations without NDA, we cannot give a lot of informations. Even above recommendations may be not totally exact. A more complete analysis of the chosen tag (with documents of manufacturers) should be done. Some general informations about current cryptographic algorithms may be found in the section 7.2.

UHF tags This report focuses on C1G2 tags. It is highly recommended to read the complete standard: [26]. This paragraph is just a short summary.

Memory bank C1G2 tags have four memory banks:

- EPC: often 96 bits (plus hidden data).
- reserved: where password are stored
- TID: contains data to identify the manufacturer and can also contain a tag serial number.
- User: is optional and can contain any data. Often 512 bits.

EPC id According to EPC C1G2 standard, EPC id shall be either as defined in ISO/IEC 15961 or as defined in the EPC Tag Data Standard ([24]). Since ISO/IEC 15961 access is not free of charge, we cannot have seen how ISO/IEC 15961 EPC ids look like. So EPC Tag Data Standard EPC ids are used (in this report) and more precisely GID-96. It avoids collision issues with tags of another company (each company has a different "General Manager Number" — this number is attributed by GS1). Since EPC is not necessarily unique, object class and serial number (60 bits) can be used as standard memory for storing aggregating data.

More precisely here is the EPC id structure:

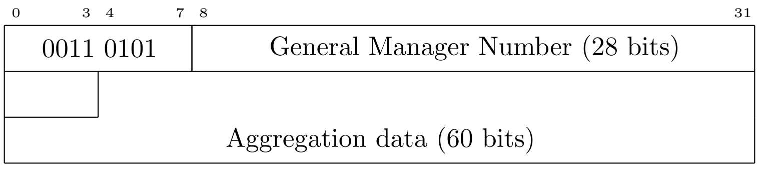


Figure 19: EPC data memory

Unique id A lot of constructors put a unique serial number (an id) in the TID. This id is factory programmed and cannot be changed by a tag interrogator. It provides a (very lightweight — see section 6.2) unique id.

Alien Technoloogy® Higgs 3 tags have a very interesting feature: dynamic authentication. Dynamic authentication enables an Alien interrogator to recognize genuine Alien tags. For more information see: [1]. Furthermore Alien ensures the TID memory contains an tag unique serial number (called UID) of 64 bits. So it provides an unique id. The security of this unique id depends a lot on the dynamic authentication protocol in particular if the serial number is taken into account by the dynamic authentication (for more information see section 6.2).

Kill password All C1G2 tags can be killed, that means can be completely disabled such that tags do not respond anymore to a tag interrogator. Killing a tag requires a kill password. If the kill password is 0, the tag cannot be killed (or more precisely, the kill password must first be changed).

The kill password can also be used (in some tags) to recommission a tag (see standard for more information).

A tag interrogator does not directly send the kill password but send the kill password XORed[25] with a data previously sent by the tag. Since it is more difficult to eavesdrop communication from tag to interrogator than the reverse, eavesdropping kill password is quite difficult.

The kill password can be used to do a password authentication as explained in [7]. It enables to implement a small protection against cloning (see section 6.6.

Access password Access password can be used to enter the secured state. By default, if access password is implemented (it is not the case with all tags) and if it is not 0, the tag is in the open state (otherwise it is in the secured state). Some features of the tags are only available when the tag is in the secured state.

As kill password, access password is not directly sent by the tag interrogator but is first XORed with a data previously sent by the tag.

Lock In the secured state, it is possible to lock memory bank or password. Locking a memory bank prevents from writing to this bank, if the tag is not in the secured state, whereas locking a password, prevents reading and writing this password, if the tag is not in the secured state.

A tag interrogator can also make permanently unchangeable the lock status of a password or memory bank (if the tag is in the secured state). If the lock status is "lock", it is said the interrogator permalocks the password or the memory bank, otherwise it permaunlocks it.

Some tags can also permalock blocks of memory (instead of complete memory bank).

Alien Higgs 3 tags also provide read lock features. Each block (64 bits) of user memory can be read locked such that it can be read only when the tag is in the secured state.

Important recommendation In aggregates based application, availability is often important and so it is highly recommended to permalock the kill password to 0. This prevents an attacker from killing or recommissioning tags.

Other special features Impinj® Monza™ 4QT tags implement QT technology: some data can only be read when the tag is near the tag interrogator. It improves data privacy because an attacker cannot easily put a tag interrogator near a tag: putting a tag near a tag interrogator is a volunteer action of the owner of the tag.

SecureRF Lime tags are battery-assisted C1G2 tags which provide authentication and encryption. According to their description, they seem unclonable (with a solution near from randomized encryption, section 6.2.1). SecureRF Lime tages use the Algebraic Eraser™ to perform asymmetric authentication. This algorithm is described in a published article [3] but the used parameters are proprietary and are not known. There are attacks on this algorithm: [35] and [29] but it is not sure these attacks work with SecureRF parameters.

[25]XOR: exclusive or.

7.2 Algorithms

This subsection focuses on practical choices of algorithms used by the controller. Tag authentication or unclonability features are not discussed here since it requires some hardware features on the tag.

Security strength As defined by NIST (National Institute of Standards and Technology) in [37], the security strength of a cryptographic algorithm or system is a number associated with the amount of work (that is, the number of operations) that is required to break a cryptographic algorithm or system. For a signature or an encryption scheme, “break” means “find the private key”. For an hash function, “break” has different meanings. In our case, “break” means “find a second-preimage” (see 3.3). Security strength is measured in bits. If 2^N execution operations of the algorithm (or system) are required to break the cryptographic algorithm, then the security strength is N bits.

Recommended security strength can be found on this website: [53] and in this NIST document: [37]. We will suppose the application will only need 80 bits security.

However this notion is not sufficient. It may actually be possible to sign documents without knowing the private key or to recover some bits of information of an encrypted text for example. Furthermore, schemes may be proven secure: it means it is possible to prove that the scheme is secure (in a very precise way, for example it may not be possible to find any bit of the encrypted text without access to an encryption or decryption system) if doing something (for example factorizing integers) is impossible.

Unfortunately such secure schemes are relatively recent and not standardized nor really used in practice (except for example RSA-OAEP which can be proved secure).

We will mainly focus on currently used algorithms and on NIST’s recommendation but when these standard algorithms are not sufficient, recent algorithms will also be presented.

Key period and attack time An important point is that keys shall be regularly changed. Security level also depends on the key periods. For example, if keys are changed all days, key size might be smaller than if keys are changed each year. Some recommended key periods are given in the website [53]

The possible time of the attack is very important too. If an attacker has less than 15 minutes to perform his attack (because the aggregated object is used only 15 minutes for instance), security mechanisms could be weaker than if the attacker has more than 1 day to perform his attack.

Side-channel attacks Even if protocols and algorithms are safe (in theory), it may be possible to break the system thanks to a side-channel attack. According to wikipedia ([51]) side-channel attack is “based on information gained from the physical implementation of a cryptosystem”.

There are a lot of different side-channel attacks: physical dismantling of a genuine tag or a genuine tag interrogator, power monitoring attack, . . .

7.2.1 Hashing functions

If the hash function used by aggregating or verifying algorithm is second preimage resistant, aggregates are protected against some attacks without need of other cryptographic mechanism, as explained in section 3.3.

In this section, some tips will be given to choose a second preimage resistant function, according to the NIST (document [18]).

We will focus on approved by NIST hash functions: SHA-1 (160 bits), SHA-224 (224 bits), SHA-256 (256 bits), SHA-384 (384 bits) and SHA-512 (512 bits). It may also be appropriate to use a subset of the bits produced by one the previous cryptographic hash functions as the (shortened) digest. If the desired digest size is λ bits, the λ left-most bits shall be selected.

The estimated second preimage strength (in bits) depends on the max message length (to be hashed) N in bits (except for SHA-384):

$$\begin{cases} \min(\lambda, 384) & \text{for SHA-384} \\ \min\left(\lambda, L - \log_2\left(\frac{N}{B}\right)\right) & \text{for other} \end{cases}$$

where L is the size of the message digest and B is the block size.

Here are a summary for non-truncated versions (where $N = 2^{16} = 65536$ bits, which is very pessimistic[26]):

	L (bits)	B (bits)	security (bits)
SHA-1	160	512	153
SHA-224	224	512	217
SHA-256	256	512	249
SHA-384	384	1024	384
SHA-512	512	1024	506

Suggestion 7.2.1. *For 80 bits security, truncated to 80 bits SHA-1 (or SHA-224) may be used.*

7.2.2 MAC and signature algorithms

MAC algorithms It seems HMAC is the most used MAC algorithm. A complete description of the algorithm can be found in [30].

Suggestion 7.2.2. *For 80 bits security, HMAC-SHA-1-80 (HMAC with SHA-1 hash function and truncated to 80 bits) with key of 160 bits can be used.*

Since verifying a MAC requires to make a request to a genuine controller (that means put a tag with the MAC near the verifying system and see if it accepts the tag or not — see section 6.4.3), if controller does not verify more than one aggregated object per second, guessing a MAC by brute force takes 2^n *seconds if MAC has n bits. Therefore, HMAC-SHA-1-32 may be sufficient since* 2^{32} *seconds are about* 136 *years.*

However there may be some physical attacks which reduces this time.

Standard signature algorithms DSA, RSA and ECDSA are the three NIST approved signature's algorithms. A complete descriptions of these algorithms can be found in [36].

Here is a summary array (according to [37]):

	Parameters[27] size (bits)	Private key size (bits)	Public key size (bits)	Signature size (bits)	Estimated security (bits)
DSA (1024, 160)	2208	160	1024	320	80
DSA (2048, 224)	4320	224	2048	448	112
DSA (2048, 256)	4320	256	2048	512	112
DSA (3072, 256)	6400	256	3072	512	128
RSA (1024)	0	1024	2048	1024	80
RSA (2048)	0	2048	4096	2048	112
RSA (3072)	0	3072	6144	3072	128
ECDSA (160)	EC[28]	160	160	320	80
ECDSA (224)	EC	224	224	448	112
ECDSA (256)	EC	256	256	512	128

Short signature algorithms Since tags often have a very limited amount of memory, shorter signature algorithms can be preferred. There are two main kinds of short signature (with size less than about 160 bits for 80 bits of security):

- multivariate signature like Quartz (128 bits - [16, 40]) or McEliece (87 bits - [15, 14])
- pairing based signature like BLS (160 bits - [11]), or three other algorithms in [10] (320 bits, 160 bits and 161 bits)

These two kinds of cryptosystems are very recent and there are not widely used. Multivariate cryptosystems have some disadvantages over pairing based cryptography:

[26]In real life, aggregated object contains rarely more than several tens of tags.
[27]Parameters can be common to all used keys.
[28]Parameters of the elliptic curve (a bit big but can be common to all keys).

- they are not studied a lot.
- public keys are often very long (1.125 Mb for McEliece and 71 kb for Quartz).
- signing time is very long (see above website for more information) and verification time for McEliece too.
- security of these cryptosystems are not fully understood according to the conclusion of this article [22] (which totally breaks the SFLASH cryptosystem which is a multivariate signature).

However there are the only current way to do very short signature (about 80 bits for McEliece cryptosystem).

Pairing based cryptography are based on pairing which are particular computable maps. Generally, pairing are done over subgroups of elliptic curves. Security strength, speed and size depend a lot on the chosen elliptic curves. Pairing are quite long to compute (see appendices D and E).

In appendix D, possible practical implementations of pairing based signature are given.

7.2.3 Cipher algorithms

Symmetric encryption The standard block cipher algorithm is AES[29]. Three versions exist: AES-128, AES-192 and AES-256. The number after "AES" is the size (in bits) of the key. The size of input and output blocks are always 128 bits. Since there is no known attacks on AES, estimated security strength of AES is the key size.

Since block cipher operates only on blocks of fixed length (128 bits for AES), block cipher modes were invented. A description of currently used modes can be found in [50].

All modes use an initialization vector which is often a 128 bits data. The initialization vector can be retrieved from plain text data and is not secret. It enables to prevent two same data from having the same cipher text.

Warning 7.2.3. *In OFB, CFB and ECB modes, if the same initialization vector is used twice, some information on data may be recorded.*

Suggestion 7.2.4. *The initialization vector should contain a random number or a timestamp in order to ensure the uniqueness. The size of this random number will be discussed later in this paragraph.*

If plain text data (to be encrypted) does not have a size multiple of 128 bits, CBC mode does not work but CBC, OFB and ECB modes works.

Warning 7.2.5. *With the OFB or ECB mode, if an attacker knows a cipher text and the corresponding plain text, he can easily encrypt any other plain text if he uses the same initialization vector. Actually, it is the case with any stream cipher (for a definition of a stream cipher, see [52]). Since, in most cases, plain text is known (it is often part of node ids and node ids are just known digest), this attack must be taken in account. Using MAC instead of hash function may solve this problem 6.4.3 because plain text cannot (easily) be known.*

Warning 7.2.6. *With CFB mode, if the plain text has $k \times 128 + n$ bits ($0 \leq k$ and $1 \leq n \leq 128$), the last n bits may be modified easily. Other bits may also be modified but modifying them will change several following bits.*

Suggestion 7.2.7. *If MACs (instead of hash functions) are not used, it is recommended to use CFB mode and to ensure that the last n bits of plain text cannot be guessed*[30]*. If it is not possible, CBC mode should be used (if data has a size which is multiple of 128 bits).*

Or it may be possible (but it is not sure) to encipher in CFB mode twice; the second time, another key (or another initialization vector — which can be the cipher text of the previous initialization vector) is used and the cipher text (ciphered one time by CFB) is reversed (left bits become right bits) and is encrypted.

[29]There are two other well known symmetric block cipher: DES and 3DES. But DES can be broken in a single days ([54]) and 3DES is less secure than AES and is slower. However if only 3DES is available, 3DES may be used.

[30]That means: an attacker shall have to try all 2^n values to find the good one and n is big enough. Another possibility is that it is impossible to modify these bits: it is the case when these bits are part of a signature, for instance.

For example, let m be the plain text, IV be the initialization vector, $e_{K,IV}$ be the AES-CFB encryption function (using the key K and the initialization vector IV), and e'_K the AES encryption function (using the key K). The twice CFB mode encryption process is:

1. *Compute $m' = e_{K,IV}(m)$.*
2. *Compute $IV' = e'_K(IV)$.*
3. *Compute $m'' = e_{K,IV'}(reverse(m'))$ where $reverse(m')$ is m' where all bits are reversed (the first bit is the last bit).*

m'' is the encrypted text. The decryption process is obvious.

This second method only works with plain text with at least 129 bits.

Modifying such a cipher text (encrypted in CBC or twice CFB mode) is more difficult, since modifying one bit change a lot of other bits.

Warning 7.2.8. *With OFB and ECB modes, it is very important to prevent an attacker from choosing its initialization vector and so encrypting any data, as soon as he knows a plain text and its corresponding cipher text. That means the initialization vector shall contain data which depends on the tag. Tag id is often a good choice, in particular when tag id is unique (see section 6.2).*

Suggestion 7.2.9. *The initialization vector can be composed by:*

- *the tag id*
- *a random number (or a timestamp)*
- *other data such version number of the aggregating algorithm or application number (UbiPark may have a different number from UbiQuitus) if such data are written in the tag (as plain text)*

If tag id is unique (see section 6.2), the size of the random number depends on the number of aggregates which can be made with the same tag. If tag is destroyed after usage, there is no need for a random number. If a tag may be used in N different aggregated objects with the same private key, random number size should be at least $\frac{1}{2}\log_2 N$ (according to birthday problem).

Authenticated symmetric encryption Two authenticated symmetric encryption modes for AES are standardized by the NIST: CCM ([21]) and GCM ([23]).

GCM has a lot of advantages, in particular, it is faster. However it requires an unique initialization vector otherwise, forgery can be made. For CCM, it is better than the initialization vector is unique but it is less critical.

Since ensuring the uniqueness of the initialization vector is quite difficult with tags (see previous paragraph), CCM is preferred.

CCM tag size (tag is the extra-data used to do the authentication, in this case) has approximately the same requirements than HMAC (see section 7.2.2).

Asymmetric encryption As seen in section 6.5, asymmetric encryption is not very useful in general for security of the whole system.

Main algorithms for asymmetric encryptions are RSA and El Gamal. However these algorithms shall not be used as is because these algorithms are very malleable (for example for RSA the product of cipher text modulo the used RSA modulus is the cipher text of the product of the corresponding plain text).

RSA encryption's estimated security strength is the same as RSA signature's estimated security strength (see page 7.2.2), whereas El Gamal's one is the same as DSA's one.

Suggestion 7.2.10. *The standard RSA-OAEP defined in PKCS #1 ([28]) may be used. This standard is proved secure under the RSA assumption (classical cryptographic assumption).*

Asymmetric encryption is not very convenient for encrypting big text because cipher text are larger than plain text and encryption and decryption are quite long operations. If large plain text have to be encrypted, hybrid encryption may be used: choose a random symmetric key (for example for AES) and encrypt this key with RSA-OAEP for example and then use this symmetric key for encrypting plain text.

Signcryption may also be a good solution when signature is also needed.

Signcryption Signcryption is a recent cryptographic primitives which simultaneously perform a signature and an encryption. Yuliang Zheng has invented signcryption in 1996. This invention were first disclosed to public in 1997 in [57].

Signcryption costs less computation time and less memory space than an encryption followed by a signature (or the reverse).

For more information, see Yulian Zheng's website [58]

7.3 Two possible implementations

As seen in the previous subsections, a lot of choices have to be made. In this section, we propose two implementations: low-cost and Higgs 3 implementation.

The low-cost implementation is really cheap: it does not use any cryptographic features on tags and the controller just needs to have widely used and fast symmetric cryptographic primitives.

The Higgs 3 implementation is based on Alien Higgs 3 tags since these tags are the only[31] passive C1G2 tags that provides dynamic authentication.

The last part of this subsection is devoted to improvements of the Higgs 3 solution.

7.3.1 Low-cost implementation

For this implementation, used C1G2 tags need to have the following features:

- a 96 bits EPC memory
- an unique (factory programmed) serial number (in the TID)
- physical features of section 7.1.2 depending on the application requirements
- an access password (optional)

There is no need for any other feature (in particular user memory is not required). Notice the required features are quite minimal.

The aggregating format can be the original format or the third tree format. The first tree format can also be used by modifying terms (remove "MAC 1" and "MAC 2" terms in the figure 20 and replace them by "aggregating data (160 bits)").

It is recommended to take account of suggestion 2.2.3.

The hash function is replaced by the following MAC algorithm (as explained in section 6.4.3): HMAC-SHA-1-24 (see section 7.2.2 for understanding this notation). The key length might be 160 bits or more.

The **first MAC** or **MAC 1** corresponds to the first digest and the **second MAC** or **MAC 2** corresponds to the second digest (for the definitions of first and second digest, see remark 3.1.2).

One problem is that they may have a lot of collisions: as soon as about $2^{12} = 4096$ MACs (a MAC correspond to a node of an aggregated object) are created, there is a collision with a high probability (because of the birthday problem). Verifying algorithms shall try a lot of possibilities to determine which tags constituted an aggregated object. But it is often possible.

However, in the UbiPark application (and maybe in a lot of applicationqs), the number of different created aggregated objects to have a collision with a high probability can be increased to about $2^{24} \approx 16 \times 10^6$. The idea is to use the original format and to put in the second MAC of the user badge the first MAC of one object of the bike. Hence an aggregated object is "defined by 48 bits of MAC", instead of only 24 bits.

The 96 bits EPC memory is organized as depicted below (GID-96 format is used — see section 7.1.4 for more informations):

0 3	4 7	8 11	12 15	16 31
0011 0101		General Manager Number (28 bits)		
	App	Ver	Key	MAC 1 (24 bits)
		MAC 2 (24 bits)		

[31] Our Internet search give us this result. However, other passive secured C1G2 can exist in the future. In particular, SecureRF has planned to produce a secured passive C1G2 tag.

Figure 20: EPC data memory for low-cost implementation

In addition to the two MACs, the EPC memory contains an application number (App — 4 bits) and a version number (Ver — 4 bits) to enable having multiple applications with different versions[32] with this implementation. A key id (Key — 4 bits) is also written in the EPC memory. The key id enables to change key used by HMAC.

Another optional protection can be added to avoid too easy cloning[33]. A HMAC-SHA-1-32 might be computed over all data of tag (including TID and EPC) and be stored as the access password. The access password can then be used to authenticate the tag (see section 6.6). If there is no access password, kill password (which is mandatory for all C1G2 tags) can be used instead (see section 7.1.4).

Remark 7.3.1. *It is highly recommended to use three different keys for the three HMAC algorithms. The key id can refer to a set of three keys.*

Warning 7.3.2. *As explained in 6.4.3, to ensure security of MACs, an attacker shall not be able to verify too many MACs per second (if he has access to a single verifying system).*

Remark 7.3.3. *It is possible to encrypt the 60 last bits of the EPC data or just the two HMAC (in this case, it is easy to use several AES keys thanks to the key id field). AES-CFB or AES-OFB (CFB and OFB are identical here since* $60 \leq 128$*) can be used. The initialization vector shall contain the tag serial number.*

One advantage of encryption is that it prevents an attacker from saying if two tags are in the same aggregated object or not (since the encrypted version of MAC 1 and MAC 2 depends on the tag identifier).

Another advantage is that it avoids two tags having the same EPC. According to the EPC C1G2 standard, reading two tags with the same EPC is not an issue. But some RFID readers seem having trouble managing tags with the same EPC.

Remark 7.3.4. *If tags are used only in one aggregated object, are programmed only once and do not need to be killed, they should be permalocked (including kill password which shall be 0). It enables to prevent any attacker from modifying tags or killing them.*

If there is an access password and if tags shall be able to be reprogrammed, they should be locked (including kill password which shall be 0).

7.3.2 Higgs 3 implementation

This implementation uses Higgs 3 tags. The security of this implementation depends a lot on the security of the Alien dynamic authentication.

Supported aggregating algorithms are the same as for the low-cost implementation except that a HMAC-SHA-1-80 is used instead of a HMAC-SHA-1-24.

Higgs 3 tags have:

- 512 bits of user memory
- 96 bits of EPC
- 96 bits of TID including 64 bits of unique identifier (called UID)

Only the user memory is used. Its organization is depicted in figure 21.

The header is not encrypted.

The cipher text part is encrypted by AES-CFB[34]. The initialization vector is depicted in figure 22.

The random number in the header (see figure 21) should be randomly chosen each time the tag is written. It enables to avoid that the same initialization vector is used twice (but due to birthday paradox, it works approximatively only when the tag is written less than $2^5 = 32$ times).

Tag data are signed thanks to a DSA (or ECDSA[35]) signature of 320 bits. More precisely, data of figure 23 are signed.

[32]The version enables to change a bit the aggregating format for example. Each verifying system shall support all versions, whereas an aggregating system can support only the last version.

[33]As explained in section 6.2, the factory programmed serial number is not sufficient since it may exist blank tags where the TID can be programmed and since, anyway, there are tag simulators. However this very light protection might be

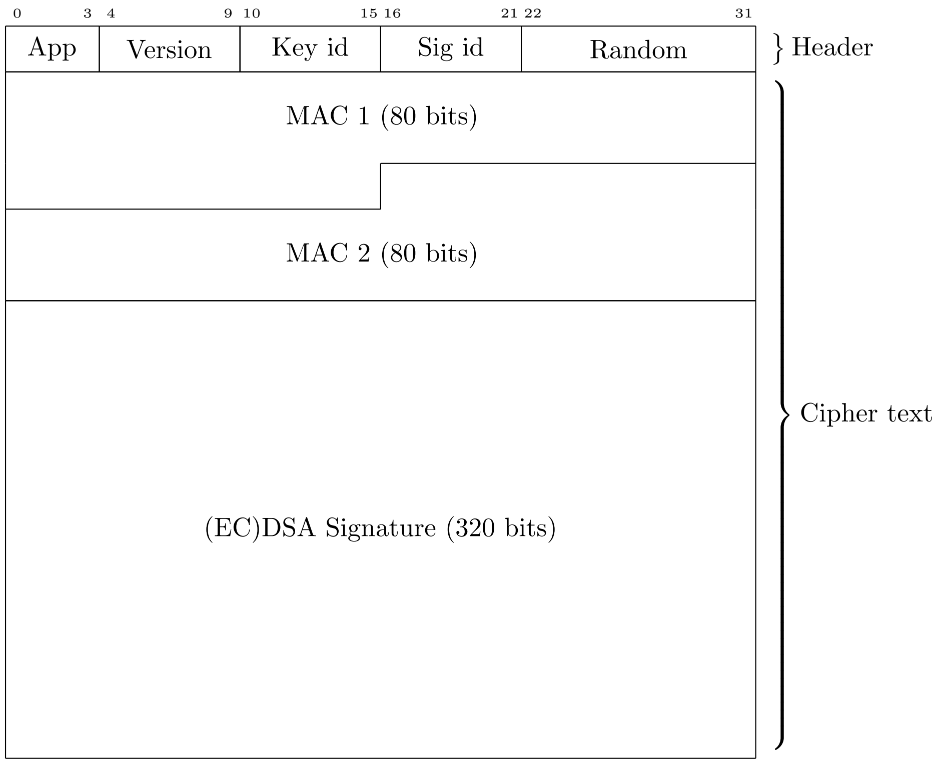


Figure 21: User data memory of the Higgs 3 implementation

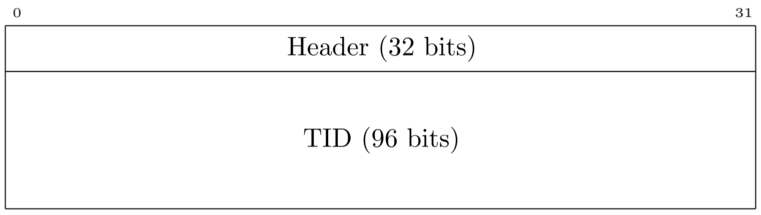


Figure 22: Initialization Vector

Each aggregating controller should have its own private key to sign tags. Each verifying controller shall have all public keys corresponding to these private keys. If an aggregating controller behaves badly (sign tags for attacker), its public key is removed from verifying controllers.

There are many ways to deal with access password. It can be:

- a MAC of the whole data of the tags (as with low-cost implementation). In this case, an authentication can be made thanks to this password (as with low-cost implementation). However if eavesdropping is possible, if availability is important and if data are not permalocked but just locked, it enables an attacker to easily write random data in tag or kill the tag (since the password is regularly sent over the air, it is easier to eavesdrop it).
- a common password. In this case, a read lock can be performed. But take also care of the above recommendation (with eavesdropping). In addition risks are higher since if an attacker get the password, it can kill all tags, not only the eavesdropped tag...
- a password stored in the user badge. It supposes the user badges is a cryptographic tag. In this case, a read lock can be performed. But take also care of the above recommendation (with eavesdropping).

sufficient when security is not very important.

[34]The plain text version of the encrypted text cannot normally be known because HMAC key and signature private key are not known by the attacker. Therefore attacks presented in warning 7.2.5 cannot be directly performed.

[35]ECDSA and DSA are very similar. Depending on the hardware, one may be faster than the other.

TID (96 bits)
EPC (96 bits)
User Data (352 first bits)

Figure 23: Signed data

Kill password recommendations are the same as for low-cost implementation (see remark 7.3.4).

Remark 7.3.5. *The EPC memory is not used. It enables the aggregating company not to buy a General Manager Number to GS1.*
However it may be useful to buy one anyway, because it enables to:

- *easily filter the interesting tags thanks to the General Manager Number. Since RFID tags will be more and more used, it might become very important.*
- *store extra data (to secure such data see section 3.3.3).*
- *store a part of the first MAC in the EPC such that user data does not need to be read to retrieve aggregated object (and then a complete verification with the user memory is performed). It can be very advantageous because reading EPC data is a lot more faster than reading user memory.*

Comparing to the low-cost implementation, this one has the following advantages:

- security strength has been improved by increasing the size of MACs.
- tags are signed. So attacks with fake tags cannot be performed. Furthermore even if an attacker can get symmetric private keys (by dismantling a genuine controller for example), it cannot easily get all private keys used for signature. And it may be possible to revoke all private keys he has theft.
- cloning protection is improved thanks to dynamic authentication (except if this mechanism is weak but it is not possible to know this because the specifications are not public).

7.3.3 Improvements and discussion

Firstly, the two implementations have been chosen because of their flexibility: authentication and signature are not mandatory to use tags. Some tags interrogator (in secure areas) can implement them, others (in less secure areas) might not implement them.

The Higgs 3 implementation is quite very secured, if Alien dynamic authentication is secured. However it is not proved…

If there were C1G2 tags with a real unique id feature (such as described in section 6.2.2), the Higgs 3 solution (applied to these tags) may be sufficient for almost all applications.

But, most aggregates based applications require each user to have a badge (for instance in UbiPark or UbiQuitus, it is the case). This badge might be a very secured HF badge (relying on widely used cryptographic primitives) to ensure a higher level of security.

The main issue is maybe the privacy: all tags have a unique id really easy to follow. A real authentication of the tag and a randomized encryption as described in section 6.2.1 is a partial solution. But according to [4], privacy (and non traceability) is a very difficult problem with RFID tags.

Anyway and in summary, in any aggregation application, two main points shall be verified (except if security has strictly no importance):

- tags shall be attached to physical objects. It shall not be possible to unstick them.
- tags shall not be clonable.

8 Conclusion and acknowledgements

This three month internship was very interesting. It enables me to know better RFID technology, apply cryptography to concrete problems and discover notions of dependability and pairing based cryptography.

In addition, I have learned Java to program benchmarks and implementations of second tree aggregating format (inside the previous Java architecture) and of a virtual tag reader, which enables to easily simulate tags and tag readers in order to test aggregating algorithms.

There still remains a lot of work to design a commercial secure product using aggregated objects, for instance UbiPark. In particular, the new global architecture (section 1.4) need to be implemented and non human-made faults (and more generally dependability impairments) should be more studied. Generalization of CMPT aggregating format can also be studied. But the most important task is maybe to do real tests with Alien tags and readers, particularly to verify if it is really possible to monitor a whole shed, despite potential perturbations due to metal of bikes.

I would like to thank the whole ACES team, in particular Michel Banâtre, Fabien Allard, Jean-François Verdonck and Paul Couderc.

The introduction of this report has been partially written by Jean-François Verdonck, intern of the École des Mines de Nantes, and some parts have been taken from Paul Couderc's articles. The global architecture (section 1.4) has been designed in coordination with Jean-François Verdonck. The idea consisting in not distinguishing left and right digests (for second tree aggregating format — see annex B.3) comes from Fabien Allard. And the analysis of dependability impairments comes out of several discussions with Michel Banâtre, Fabien Allard and Jean-François Verdonck.

Appendices

A Another generic verifying algorithm

A.1 The first verifying algorithm

The first generic verifying algorithm (algorithm A.1.0.1) takes the set of tags seen by the tag interrogator and a tag. It returns the maximum tree containing this tag.

Algorithm A.1.0.1 First generic verifying algorithm: $check1(S, u, h')$

```
Require: $S$ is the set of tags seen by the tag interrogator.
Require: $u$ is a tag (leaf) in $S$.
Require: $h'$ is an id (or more precisely a digest) or $null$ (see below)
Ensure: return maximum tree with $u$ whose the father of the root has digest $h'$ if $h' \neq null$, otherwise
  return maximum tree with $u$.
  $T \leftarrow \{u\}$ the current tree (aggregated object or tag)
  $v \leftarrow u$ the current root
  while one tag (leaf) of $T$ contains the digest of the father of $v$ the root of $G$ do
    $h \leftarrow$ the digest
    if $h = h'$ then
      return $T$
    end if
    $B \leftarrow \{T\}$ (the children of the node $h$)
    for $w$ tag descendant of a node of id $h$, which is not in any tree of $B$ do
      $S' \leftarrow S$ without tags in any tree of $B$
      $T' \leftarrow$ check$(S', w, h)$
      if there is an error then
        exit the "while loop"
      else
        $B \leftarrow B \cup \{T'\}$
      end if
    end for
    if a subset $B'$ of $B$ can be an aggregated object (i.e. digest is correct) then
      $G \leftarrow$ tree aggregated object corresponding to $B'$
      $v \leftarrow$ the root of $T$ (a node with id $h$)
    else
      exit the "while loop"
    end if
  end while
  if $h' = null$ then
    return $T$
  else
    return error
  end if
```

To use this algorithm with a given aggregating format, it shall be indicated:

- how the parent id of a tree T can be computed thanks to the data in the tags of T (for the "while loop").
- for each children subtree of a given node u, how to find a tag which is in this subtree, when known information are the digest of the node u (for the "for loop"), if all tags of aggregated object of root u are in S (i.e. are seen by tag interrogator). For example in the tree of the figure 6, given the tags a, b, c, d and e, and given h_3, it should be possible either to say that a is a descendant of h_3 or to say that b is a descendant of h_3. And idem with c, d and e (instead of a and b).

The first requirement is the same as the requirement of the second verifying algorithm (algorithm 2.3.2.1).

The second requirement is always verified when, in any subtree T of an aggregated object, there is a tag which contains the id of the root of T. This is the case with all formats except first tree tree format. For the first tree format, the first generic verifying algorithm cannot be used as is. There is no such issue with the second verifying algorithm.

Example A.1.1. *Suppose the aggregated object of the figure 6 has been created by a good aggregating algorithms (the colored arrows are not the one written on the figure).*

Suppose the tag interrogator sees tags a, b, c, d and e but not f. $S = \{a, b, c, d, e\}$. Let randomly choose one tag of S, for example $u = d$.

When $check1(S, u, null)$ is executed:

1. *it first computes the parent id h_2 of d. This computation is made thanks to data stored in d (and depends on the tree formats).*
2. *it finds all tags whose the parent id is h_2: d and e.*
3. *it verifies the tree $T = \{c, d, e\}$ has a root digest equal to h_2.*
4. *it finds the parent id h_3 of the tree T. This computation is made thanks to data stored in c, d, e (and depends on the tree formats).*
5. *it finds a tag descendant of h_3: for example a.*
6. *it recursively call $check1(\{a, b\}, a, h_3)$ which returns the tree $\{a, b\}$.*
7. *possible children to h_3 are $\{a, b\}$ and T ($B = \{\{a, b\}, T\}$) since there are no other tags in S.*
8. *it verifies the tree $T' = \{\{a, b\}, T\}$ has a root digest equal to h_3.*
9. *it returns T'.*

Remark A.1.2. *Sometimes it may be easier to use the number of edges between u and the node with digest h' instead of directly using h' because some aggregating format store this value.*

Remark A.1.3. *The last part of the "while loop" (which tries to find a subset of B whose the digest is correct) may be skipped and $B' = B$ can be chosen. But doing this can enable an attacker to create fake tags*[36] *with already used digest in order to disturb the system (and even without attacker, collisions — two identical digests — may arrive*[37]*).*

To speed up this search of subset without enabling such attacks, number of children of each node may be stored. The problem is to decide in which tag is store which number of children. If the second requirement (see above, before the example) is verified, the number of children of a node u may be stored in all tags which are found as descendant of u.

If all tags do not have the same number of children, verifying algorithm can group them by this number.

However, storing this number of children bound the number of authorized children (since the memory size is bounded). In practice, this bound is often not a problem because aggregated objects does not contain a huge amount of tags.

A.2 Comparison with the second verifying algorithm

The first generic verifying algorithms may be practical if the verifying system shall verify the integrity of an aggregated object which contains a specific tag (for example, in UbiPark, this specific tag is the tag of the owner of the bike).

However it is often required to find all maximum aggregated objects. It is possible to use the algorithm A.1.0.1 with algorithm A.2.0.2, for that purpose.

But, in this case, the second verifying (algorithm 2.3.2.1) algorithm is more adapted.

[36] Tag authentication (see section 6.4.1) prevents the creation of fake tags. When tag authentication is used, the last part of the "while loop" can so be skipped

[37] Collisions are quite impossible with secure cryptographic hash functions.

Algorithm A.2.0.2 Second kind verifying algorithm (using *check*1 algorithm A.1.0.1): checkAll1(S)

```
Require: $S$ is the set of tags seen by the tag interrogator.
  $E \leftarrow \emptyset$ the set of maximum aggregated objects
  while $S \neq \emptyset$ do
    $u \leftarrow$ a randomly chosen tag of $S$
    $T \leftarrow check(S, u, null)$
    $E \leftarrow E \cup T$
    $S \leftarrow S \setminus \{$tags of $T\}$
  end while
  return $E$
```

B Other aggregating formats

B.1 Original format

In the original format, each tag stores the id of its father and of its grandfather. The verifying algorithms A.1.0.1 and 2.3.2.1 can be used:

- the id of the father of a tag is directly stored in the tag
- the id of the father of an aggregated object with at least one child which is a tag is the grandfather's id of this tag
- the id of the father of an aggregated object without tag child cannot be directly recovered

The last point shows the limitation of this original format. In the figure 24, the verifying algorithm cannot check the full tree because h_4 is not stored in any leave of the subtree of root h_3. It is so not possible to make the link between the h_3 subtree and f (except by trying all possible trees).

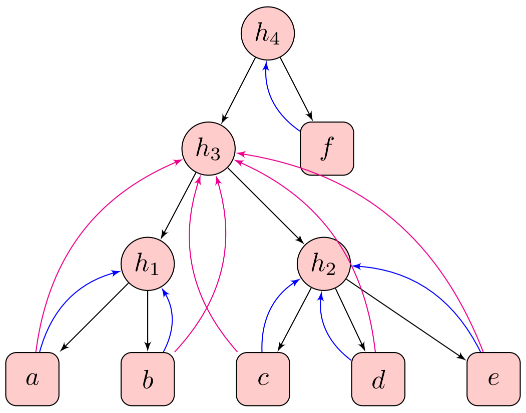


Figure 24: Original aggregating format example

B.2 First tree format

B.2.1 Basic version

In this format, each tag stores the complete id of its father, the half of its grandfather's id, the quarter of its great-grandfather's id, etc, such that for each subtree of root u, it is possible to compute the id of the father of u (if it exists). The figure 25 gives an example.

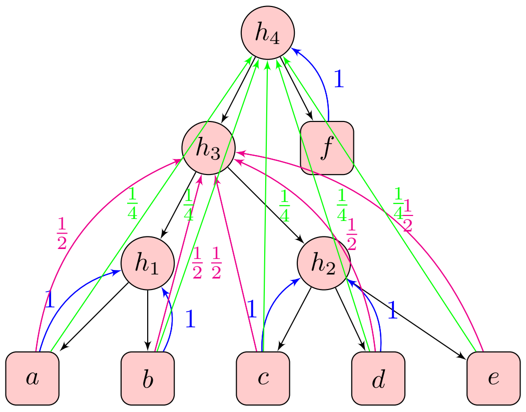


Figure 25: First tree format example

The algorithm B.2.1.1 is a possible aggregating algorithm. The algorithm uses the following notation: if h is a bit string of L bits, $h[a, b]$ is the bit string formed by h from index a (index a included, $0 \leq a \leq L-1$) to index b (index b excluded, $a+1 \leq b \leq L+1$)). If $b > L+1$, $h[a, b] = h[a, L+1]$ and if $a \geq L$ or $a \geq b$, $h[a, b]$ is empty. For example, if $h = 01001$, $h[1, 3] = 10$ (index 0 corresponds to the left bit).

Algorithm B.2.1.1 First tree aggregating algorithm: *aggregate*(E)

```
**Require:** $E$ is a set of trees to be aggregated (as explained in section 2.3)
**Require:** each tree of $E$ is a correct aggregated object. In particular each tree has at least $2L$ free bits.
**Ensure:** return aggregated object whose children of the root are elements of $E$
  $L \leftarrow$ the sorted (by id) list with the elements of $E$
  $L_{id} \leftarrow$ the related list of ids
  $h \leftarrow$ the digest $H(L_{id})$ (see section 2.1)
  **for all** tree $T \in E$ **do**
    Let number leaves of $T$ in the order given by the tree (the order of the leaves when the tree is drawn): $1, \ldots, k$
    $n_i \leftarrow \frac{1}{2^{-d_i}}$ the number of bits to be written in the leaf $i$ ($d_i$ is the depth of $i$)
    **for** $i = i$ to $k$ **do**
      Write $h[n_1 + \cdots + n_{i-1}, n_1 + \cdots + n_i]$ in free bits of tag $i$ (write the bits at left). The first written bit in the tag has index $L + \frac{L}{2} + \ldots \frac{L}{2^{d_i - 1}}$ if $d_i \neq 0$ or 0 if $d_i = 0$.
    **end for**
  **end for**
  **return** the aggregated object which corresponds to $L$
```

The verifying algorithm 2.3.2.1 can be used but not the verifying algorithm A.1.0.1. The way to find digest of the father of any aggregated object or tag is obvious.

Remark B.2.1. *In the aggregating algorithm, $n_1 + \cdots + n_k \geq L$. There is an equality if and only if the tree is binary.*

Remark B.2.2. *An other distribution of bits may be obtained by considering each node has only two children: the left child and the right child. Other children memory is not used but other children ids are taken in account in the hash tree.*

The main advantage is that it enables to spread more data bits in the tags.

If bits could be cut in two or more, since $\sum_{i=0}^{\infty} \frac{1}{2^n} = 2$, each tag need to store less than $2L$ bits where L is the number of bits of the hash function. In practice, since it is not possible to cut bits, it is not possible to represent all trees: the height is limited to $1 + \log_2 L$ if L is a power of 2. For instance, if the output of the hash function has 256 bits, the limit is 9.

B.2.2 Enhanced version

The enhanced version removes this limitation. The idea of this algorithm is to group all tags where the number of bits to be written is not integer and to consider this group as one tag. So there is no more non integer number of bits.

Algorithm B.2.2.1 is the "forall loop" (of algorithm B.2.1.1) for the enhanced version.

Other algorithms may exist to better distribute these bits.

B.2.3 Case where L is not a power of 2

When L is not a power of 2, enhanced algorithm can be used. But if L is odd, the algorithm become the third tree format (section 3.2). So the spread property described in section 3.3.4 is no more verified and this first format has no interest.

There is a trick to avoid this. Let write L in binary: $L = a_0 + a_1 2^1 + \cdots + a_k 2^k$. Let $0 \leq i_1 < \cdots < i_j \leq k$ be the index i such that $a_i = 1$. h can be cut in j parts $h_1, \ldots, h_j$ of size: $2^{i_1}, \ldots, 2^{i_k}$. So the previous enhanced algorithm can be applied (independently) to $h_1, \ldots, h_j$.

Algorithm B.2.2.1 Enhanced “forall loop” for first tree aggregating algorithm: $aggregate(E)$

```
for all tree $T \in E$ do
  Let number the leaves of $T$ whose the depth $d$ is such that $\frac{1}{2^{-d}}$ is not integer in the order given by the tree: $1, \dots, l$
  Let number the other leaves in the same order: $l+1, \dots, k$
  $n_i \leftarrow \frac{1}{2^{-d_i}}$ the number of bits to be written in the leaf $i$ ($d_i$ is the depth of $i$)
  Since $n_1 + \dots + n_k$ and $n_{l+1}, \dots, n_k$ are integer, $n_1 + \dots + n_l$ is integer.
  Write $h[0, n_1 + \dots + n_l]$ in free bits of tags $1, \dots, l$ respectively. For example, if $l = 3$ and $1, 2, 3$ have $2, 1, 3$ free bits and $n_1 = 1.5, n_2 = 1.25, n_3 = 1.25$ ($n_1 + n_2 + n_3 = 3$, $h[0, 2]$ is written in 1, $h[2, 3]$ is written in 2 and nothing is written in 3.
  for $i = l + 1$ to $k$ do
    Write $h[n_1 + \dots + n_{i-1}, n_1 + \dots + n_i]$ in free bits of tag $i$ (write the bits at left). The first written bit in the tag has index $L + \frac{L}{2} + \dots \frac{L}{2^{d_i - 1}}$ if $d_i \neq 0$ or 0 if $d_i = 0$.
  end for
end for
```

B.3 Second tree format

Several versions of this algorithm exist. The last one is the preferred one but previous versions are very useful to understand the aggregating mechanism.

B.3.1 With left/right distinction

With this format:

- the left digest is the id of the deepest (with the biggest depth — equivalent to the nearest ancestor to the tag) ancestor (father, grandfather, great-grandfather, etc) whose a right child is an ancestor of the tag, if it exists (this ancestor is called the **left ancestor** and a red arrow is used)
- the right digest is the id of the deepest (with the biggest depth — equivalent to the nearest ancestor to the tag) ancestor (father, grandfather, great-grandfather, etc) whose a left child is an ancestor of the tag, if it exists (this ancestor is called the **right ancestor** and a green arrow is used)

If the tree were a binary search tree (without two equals elements), the left ancestor would be the biggest node less than the tag and the right ancestor would be the smallest node greater than the tag.

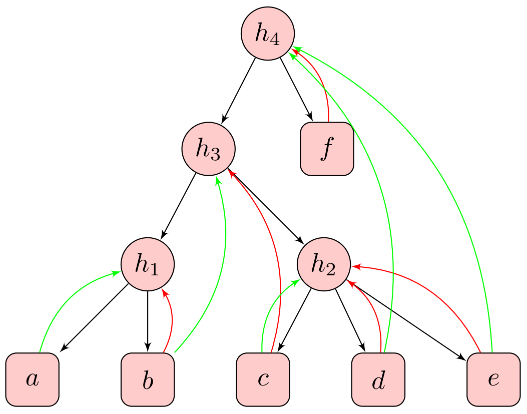


Figure 26: Second tree format (with left/right distinction) example

A first property can be seen: for each subtree, at least two tags contain the id of the subtree root. These two leaves are the leftmost leaf of the right child of the root and the rightmost leaf of the left child of the root. If tree is not a binary tree, there may be more tags which contain the id of the root: they will be called either the leftmost leaves (or tags) of the right child or the rightmost leaves (or tags) of the left child.

This is not enough to easily recover the tree from the tags (even if it may be possible). For instance, in the figure 26, the verifying algorithm cannot say if h_2 is the sibling of h_1 or of f (if $h_2 > f$) without computing the digest.

A solution can consist in storing the numbers of edges (minus 1 for space improvement) between the tag and the left ancestor (and the right ancestor — these two numbers are called the **relative depths**). If the memory space allowed to each of these numbers is k bits, all trees of depth less than equal to $2^k - 1$ are supported by aggregating format. Other trees may be not supported (but may also be supported depending of the tree structure).

The verifying algorithms A.1.0.1 and 2.3.2.1 can be used. Thanks to relative depth, it is possible to find the digest of the father of any tree just by analyzing all tags of the tree.

The algorithm B.3.1.1 is a possible aggregating algorithm. The used convention is that left child is the child with the lowest id and other children are right children.

Algorithm B.3.1.1 Aggregating algorithm for the second tree format with left/right distinction: *aggregate*(E)

```
Require: $E$ is a set of trees to be aggregated (as explained in section 2.3)
Require: each tree of $E$ is a correct aggregated object, in particular the left digest of the leftmost tags
    and the right digest of the rightmost tags are not used.
Ensure: return aggregated object whose children of the root are elements of $E$
  $L \leftarrow$ the sorted (by id) list with the elements of $E$
  $L_{id} \leftarrow$ the related list of ids
  $h \leftarrow$ the digest $H(L_{id})$ (see section 2.1)
  Write the digest $h$ in right digest of rightmost tags of the first element of $L$.
  Write the digest $h$ in left digest of leftmost tags of the other elements of $L$.
  return the aggregated object which corresponds to $L$
```

B.3.2 Without left/right distinction

It can be noticed, one of the stored digest in each tag is the father's id. The relative depth is so 1. Hence for memory space improvement, the first digest (formerly the left digest for example) of each tag contains the father's digest and the second digest (formerly the right digest for example) the left or the right ancestor which is not the father.

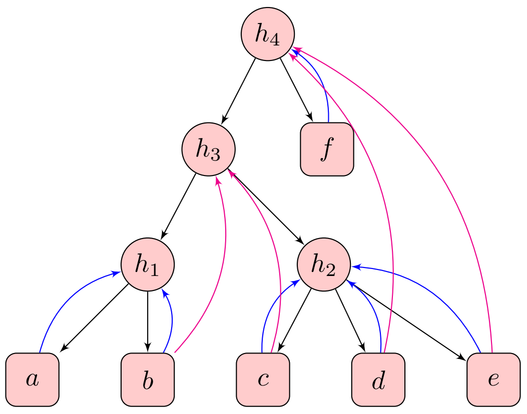


Figure 27: Second tree format (without left/right distinction) example

Aggregating and verifying algorithms are nearly the same as with left/right distinction.

C Multiple parents tree (MPT) and centered multiple parents tree (CMPT)

This section formalizes concepts of the subsection 4.1. It is highly recommended to read this subsection in order to have an intuitive idea of MPT and CMPT.

C.1 Definitions and first propositions

To simplify notations, graph vertices are considered to be natural integer (element of $\mathbb{N}$). So the set of all directed graphs can be defined. Let $\mathcal{G}$ denote this set.

Définition C.1.1. *A **simple directed graph** is an ordered $G = (V, E)$ of:*

- *a finite subset V of $\mathbb{N}$, whose are called **vertices**.*
- *a set E of ordered pairs of two different vertices, called **edges** (E is a subset of V^2).*

In this annex, simple directed graphs are simply called **directed graphs** or **digraphs**.

Définition C.1.2. *A **simple undirected graph** is an ordered $G = (V, E)$ of:*

- *a finite subset V of $\mathbb{N}$, whose are called **vertices**.*
- *a set E of (unordered) pairs of two different vertices, called **edges**.*

In this section, simple undirected graphs are simply called **undirected graphs**.

Définition C.1.3. *For each directed graph $G = (V, E)$, an undirected graph $G' = (V, E')$, where $E' = \{\{u, v\} | (u, v) \in E\}$, can be associated.*

*The graph G' is called the **undirected graph related to** the graph G.*

Définition C.1.4. *A **MPT** is a directed graph such that the related undirected graph is **connected** and **acyclic**.*

Contrary to appearances, this definition is very strong. There are very few graphs which are MPTs. For instance, the figure 28 is not a MPT.

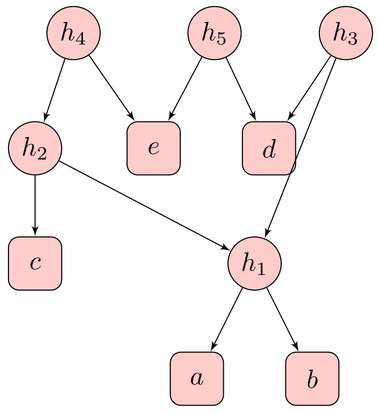


Figure 28: MPT counterexample ($h_1, h_2, h_4, e, h_5, d, h_3$ is a cycle in the related undirected graph)

Définition C.1.5. *Let consider a MPT:*

- *a **leaf** (or a **sink**) is a vertex with 0 out-degree.*
- *an **internal node** or simply a **node**[38] is a vertex with a non-zero out-degree. Vertices are partitioned into leaves and nodes.*

[38]WARNING: this is not a standard convention. Generally a node is any vertex. But in order to be coherent with tree vocabulary used in the other parts of this report, *a vertex is not (necessarily) a node.*

- *a **root** (or a **source**) is a vertex with a 0 in-degree. Since the related undirected graph is connected, if there are at least* 2 *vertex, all roots are nodes.*
- *the **parents** (or **fathers**) of a vertex u are the vertex v (or nodes) such that (v, u) is an edge.*
- *a vertex u is an ancestor of a vertex v is there is a path from u to v.*
- *the **children** of a vertex u are the vertex v such that (u, v) is an edge.*
- *a vertex u is an descendant of a vertex v is there is a path from v to u.*
- *two vertex are **sibling** if they have a common parent.*

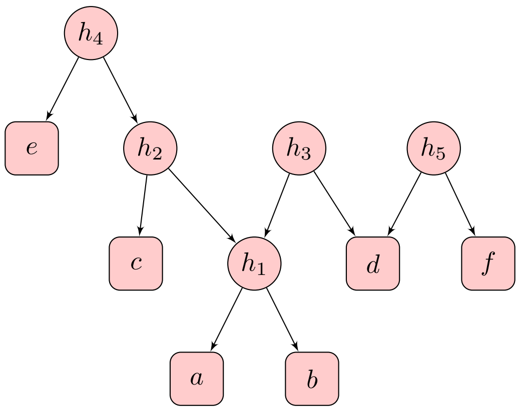


Figure 29: MPT example (a, b, c, d, e and f are the leaves; h_4, h_3 and h_5 are the roots)

Définition C.1.6. *A MPT is **binary** if each node has always* 2 *children.*

Remark C.1.7. *A **classical tree** (or simply **tree** can be seen as a MPT where each vertex has only one parent.*

Définition C.1.8. *The subtree of root u of a MPT is the classical tree constituted by u and all its descendants.*

The goal is to find an aggregating format (leaves are tags with data and nodes are virtual) such that, when a set S of tags (leafs) are read, the maximal subtrees (for the inclusion), whose leaves are in S, are detected. Actually this is the same aim as for tree aggregating format except that a node or a leaf can have several parents. Unfortunately, it seems difficult to find such aggregating format. That is why CMPT are introduced in the subsection C.5.

C.2 Generation

Let consider a MPT G.

Remark C.2.1. *Since the undirected graph related to G is acyclic and connected, for all vertex u and v, there is a unique path (in this graph) between u and v. This path is called the **undirected path** between u and v.*

Définition C.2.2. *Let u be a (arbitrary) chosen vertex and d an (arbitrary) integer (in $\mathbb{Z}$).*

The generation $g(v)$ of any vertex v is recursively defined as follow:

- *if $u = v$, $g(u) = g(v) = d$.*
- *if $u \neq v$, let v' be the vertex just before v in the undirected path between u and v. $g(v) = g(v') + 1$ if v is a child of v' and $g(v) = g(v') - 1$ otherwise.*

Proposition C.2.3. *When the reference vertex u is changed or when d is changed, the generations are just shifted. More precisely, if, for each vertex v, $g(v)$ was the old generation of the node v and $g'(v)$ was the new one: there exists an integer a such that for each vertex v, $g'(v) = g(v) + a$.*

Therefore, the following definition can be expressed:

Définition C.2.4. *Two vertices u and v are* ***of the same generation*** *($u \sim v$) if and only if $g(u) = g(v)$.*

Remark C.2.5. *The above definition is correct because the equality $g(u) = g(v)$ does not depend on the chosen reference node for computing $g(u)$.*

Proposition C.2.6. *The relation $\sim$ is an equivalence relation.*

Here is an interesting proposition.

Proposition C.2.7. *Let consider a generation H of G as a set of vertices of an undirected graph. $\{u, v\}$ is an edge of this graph if and only if u and v have a common ancestor.*

The obtained graph is acyclic (but not necessarily connected) and has at most $|H| - 1$ edges ($|H|$ is the number of vertices of the generation).

Proof. The fact that the graph is acyclic is obvious: because the undirected graph related to the MPT is acyclic.

For the edges numbers, delete all vertices with a degree less or equal to 1 (recursively such that there is no more vertex with a degree less or equal to 1). There are no more vertex.

Otherwise, let u_0 be a remaining vertex, we follow one of its edge, we arrive on a vertex u_1 with degree at least 2, we follow an edge different from $\{u_0, u_1\}$, we arrive on a vertex u_2, ... Since the number of edges is finite, there is a cycle. This is impossible. So there were no more vertex.

Furthermore, in the previous algorithm, before deleting the last vertex, there were no edge, and each time a vertex was deleted, at most one edge was delete. So the number of edges is less or equal to the number of vertices ($|H|$) minus 1. □

The next subsection is a presentation of the sub-MPT. *This is very different from the subtrees.*

C.3 Sub-MPT

Let consider a MPT G.

Définition C.3.1. *The sub-MPT of root u is directed subgraph induced by the undirected connected component of all vertex of generation greater or equal than $g(u)$, which contains u.*

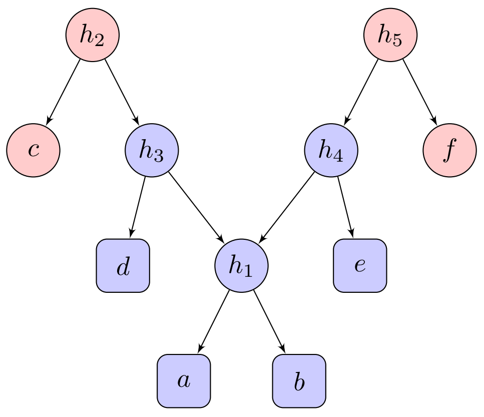


Figure 30: Example of sub-MPT (in blue)

Remark C.3.2. *A sub-MPT is a MPT.*

C.4 Recursive construction

The previous formal definition is convenient but does not enable an easy construction of MPTs. Here is a recursive definition of MPTs.

Theorem C.4.1. *The set of the MPTs is the smallest (for the inclusion) set E of graphs such that:*

- $\forall n \in \mathbb{N}$, $G = (\{n\}, \emptyset) \in E$, *that means graphs with only one vertex and without edges are in E.*
- *if:*
  1. $G_1 = (V_1, E_1), \ldots, G_k = (V_k, E_k)$ *are in E ($k \geq 1$) ;*
  2. $V_i \cap V_j = \emptyset$ *for all $i \neq j$*
  3. $w \in \mathbb{N} \setminus \left(\bigcup_{i=1}^{k} V_i\right)$ *;*
  4. $u_1, \ldots, u_k$ *are roots of* $G_1, \ldots, G_k$ *respectively,*

  then the graph $G' = \left(\bigcup_{i=1}^{k} V_i,\ \bigcup_{i=1}^{k} E_i \cup \{(w, u_1), \ldots, (w, u_k)\}\right)$ *is in E.*

The figure 31 shows the second point of the theorem: G_1, G_2 et G_3 are three graphs "linked" by w.

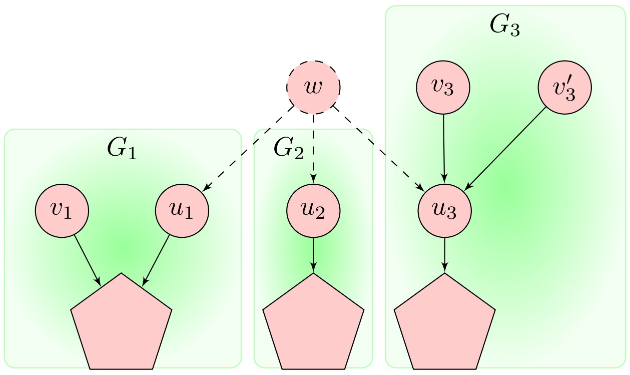


Figure 31: Illustration of the second point of the recursive definition of MPTs second point of the theorem — the dashed things are things added by the construction, the pink pentagons are MPTs)

The proof is theoretical even if the propriety is intuitively clear.

Proof. Firstly it easy to prove (by induction) that the set E (as defined above) is included in the set of all MPTs.

Now let prove the converse. Let $G = (V, E)$ be a MPT. Let number its generations such that the minimum generation is 0. Let N be the maximum generation. Let show that, for all n ($0 \leq n \leq N$), subgraphs of G induced by all undirected connected components induced by the vertices of generation greater or equal than n are in E. This proof is done by induction (from $n = N$ to $n = 0$):

- if $n = N$, the only connected components are leafs (which are subgraphs constituted by one vertex and without edge and which are therefore in E).
- Suppose the property is true for $n + 1 \geq N$ and prove it for n. The new (related to the case $n + 1$) connected components are:
  - either leaves (of generation n) which are in E.
  - or an union of old connected components $V_1 \subset V, \ldots, V_i \subset V$ induced by vertices of generations $k \geq n+1$ and nodes $w_1, \ldots, w_j$ of generation n. Furthermore the induced edges are only edges between a node w_m and a vertex u_l (in one of the $V_{i'}$) of generation $n + 1$ (by definition of generations). These edges form a subgraph $\tilde{G}$ (of the MPT), whose the related undirected graph is acyclic. So it is possible to use (many times — one time for each $w_{i'}$) the second point of the theorem, to construct the given connected component (and to prove it is in E). A formal proof can be done by induction (for each $w_{i'}$).

□

This proof is really interesting to better understand the MPTs. In particular, it enables to see that it is possible to create the MPTs generation by generation (exactly as the creation of a tree aggregated object by a tree aggregating algorithm — section 2.3).

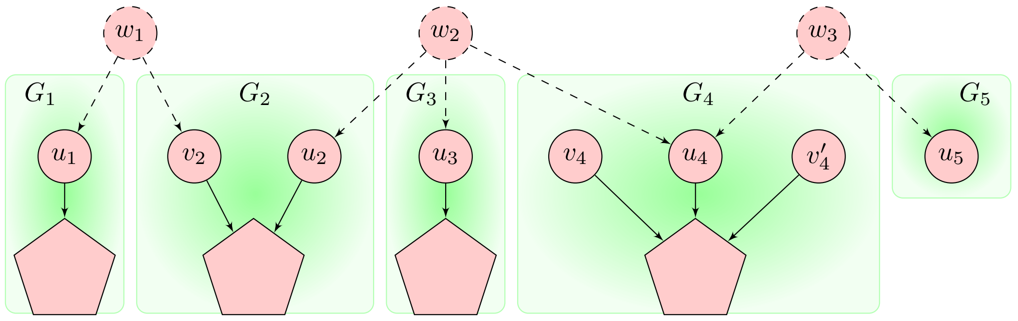


Figure 32: Illustration of the recursive construction of one generation (the graph $\tilde{G}$ corresponds to the subgraph induced by the dashed edges)

In the construction of the proof, it is very important to remark that only vertices of the smallest generation of graphs $G_1, \dots, G_i$ (related to $V_1, \dots, V_i$) can be linked thanks to a new vertex w_k.

Let denote $\hat{G}$ the graph built from $\tilde{G}$ by merging nodes belonging to the same G_i (the resulting vertex will be called G_i).

Unfortunately the MPTs are too generic and it seems very difficult to find an aggregating format for them. That is why, the study is reduced to particular MPTs: the centered multiple parents tree (CMPTs).

C.5 Centered multiple parents tree (CMPT)

Définition C.5.1. *A MPT G is a CMPT if and only if, for all sub-MPT G', there exists a vertex u in G' of generation 1 (when the minimum generation is 0) such that any node $v \neq u$ of generation 1 is a sibling of u.*

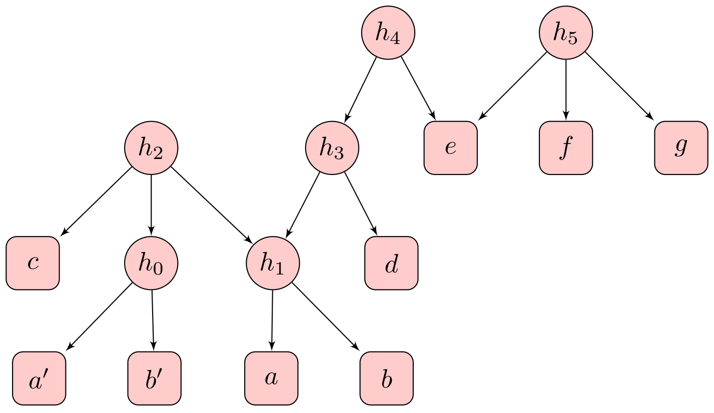


Figure 33: Example of CMPT

Another equivalent (and maybe easier to understand) definition is:

Définition C.5.2. *A MPT G is a CMPT if and only if, for all generation, at most one vertex has more than one parent.*

Remark C.5.3. *A recursive definition is also possible. Maybe, the most intuitive way to do this consists in starting from the recursive construction used in the proof of the theorem C.4.1 (see page 59) and in*

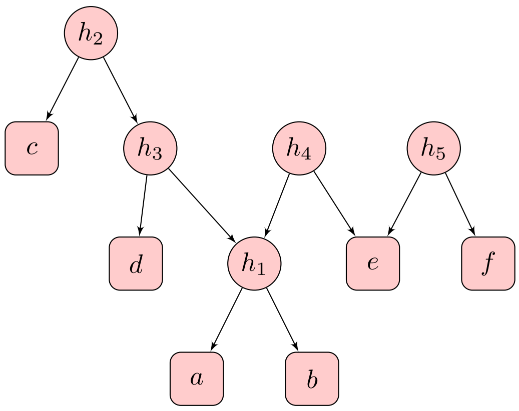


Figure 34: Counterexample (a MPT which is not a CMPT) because of the sub-CMPT of root h_3

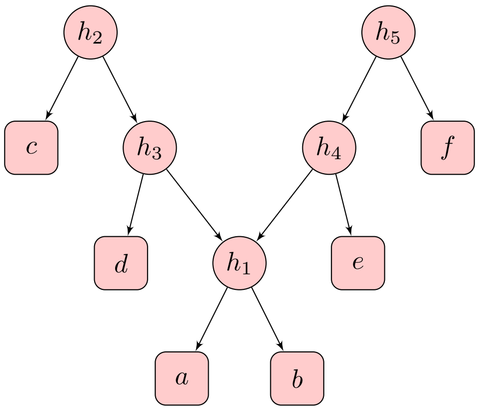


Figure 35: Counterexample (a MPT which is not a CMPT) because h_3 and h_4 are not sibling.

forcing that the built MPT is such that: all nodes of generation 1 *(if the minimum generation is* 0*) are sibling of a certain node* u*.*

Another way to see this last condition (when each node has only two children — it is easy to consider other cases but the notations become too ugly) is that the recursive construction takes $n+1$ *CMPTs* $(G, G_1, \ldots, G_n)$ *and* $n+1$ *nodes of minimum generation* $u, u_1, \ldots, u_n$ *in* $G, G_1, \ldots, G_n$ *respectively; and builds a CMPT with* n *new nodes* $w_1, \ldots, w_n$ *and* $2n$ *new edges:*

$$(w_1, u_1), (w_1, u), \ldots, (w_n, u_n), (w_n, u)$$

u *and* G *can be considered as in the* ***center*** *of the built CMPT.*

This recursive construction is the one described in the section 4. The figure 12 (page 17) illustrates it.

D Practical implementations of pairing based signatures

As explained in section 7.2.2, a lot short signatures schemes use pairings on elliptic curves.

This whole annex is based on the thesis of Ben Lynn [32].

We suppose the reader has some basic knowledge in elliptic curves. In addition this annex is not very detailed and reading the thesis is highly recommended.

In the first subsection, a very short introduction to BLS signature is given. The main goal of this introduction is to explain why pairings are very useful; or more precisely to introduce a problem (finding a co-GDH group), which can be solved thanks to pairings.

In the second subsection, pairings are shortly described and in the third subsection, some possible parameters are given and compared. If you want to use a short pairing-based signature, reading this third subsection may be sufficient.

D.1 Introduction to BLS signature

D.1.1 Discrete Log Problem, Computational Diffie-Hellman Problem and Decisional Diffie-Hellman Problem

Let G be a cyclic group of prime order r. Let g be a generator of G and let x, y, z be integers in $[\![0, r-1]\!]$.

Let define the following problems:

- **Discrete Log Problem (DL)**: given g, g^x, compute x.
- **Computationable Diffie-Hellman Problem (CDH)**: given g, g^x, g^y, compute g^{xy}.
- **Decisional Diffie-Hellman Problem (DDH)**: given g, g^x, g^y, g^z, determine if $xy = z$. If $xy = z$, the tuple (g, g^x, g^y, g^z) is a **Diffie-Hellman tuple**.

Obviously, if the DL problem can be easily solved, the two others can also be easily solved; and if the CDH problem can be easily solved, the DDH can be easily solved.

A lot of classical cryptosystems (such as Diffie-Hellman key-exchange protocol or Schnorr zero-knowledge proof) are based on the assumption one of these problems (the CDH in the first case and the DL in the second case) is difficult to solve.

Some recent cryptosystems, like BLS signature, need groups where CDH is difficult but DDH is easy. These groups are called **Gap Diffie-Hellman (GDH)** groups.

D.1.2 BLS signature (particular case)

The BLS signature is a short[39] signature scheme using a GDH group.

Let $H : \{0,1\}^* \to G$ be a hash function. $\{0,1\}^*$ is the set of all binary messages (or number). Security of H is very important: see [11], for more information.

Here is a particular case of the BLS signature scheme with GDH groups (this case is described in the thesis [32]):

- **Setup:** Choose a GDH group G of prime order r. Publish a generator $g \in G$ (g is just a random element of $G \setminus \{1\}$ the order of G is a prime number r). The group G and the generator g can be used by all users.
- **Key generation:** Choose a random $x \in [\![1, r-1]\!]$. Output the public key g^x and the private key x. This operation shall be done by any user who wants to sign messages.
- **Signing:** The signature of a message $m \in \{0,1\}^*$ is $\sigma = h^x$ where $h = H(m)$.
- **Verify:** Given a message-signature pair (m, σ) and the public key g^x, check that (g, h, g^x, σ) is a Diffie-Hellman tuple, where $h = H(m)$.

[39]Actually the size of the signature depends on the chosen GDH group. Finding a good GDH group is quit hard.

If the signature is correctly issued, the verification passes. Indeed, let write $h = g^y$ (this is possible because g is a generator of G). Then $\sigma = h^x = g^{xy}$. Thus (g, h, g^x, σ) is a Diffie-Hellman tuple.

It can be proven this scheme is secure against existential forgery under a chosen-message attack in the random oracle model (the described scheme is a particular case of the BLS signature and the proof of the security of the general case can be found in [11]).

Unfortunately, it is quite difficult to find such a GDH group, such that its elements (and so the signature) can be represented by a short binary string. But it is possible to use generalizations of GDH groups, which are easier to find.

D.1.3 Co-GDH groups

Let G_1 and G_2 be two cyclic groups of prime order r. Let g_1 be a generator of G_1 and g_2 be a generator of G_2. We suppose there is an efficiently computable isomorphism[40]: $\varphi : G_2 \to G_1$ with $\varphi(g_2) = g_1$.

The CDH and DDH problems can be generalized:

- **Computational co-Diffie-Hellman Problem (co-CDH)**: given $g_2, g_2^x \in G_2$ and $h \in G_1$, compute h^x.
- **Decisional co-Diffie-Hellman Problem (co-DDH)**: given $g_2, g_2^x \in G_2$ and $h, h^z \in G_1$, determine if $x = z$. If $x = z$, the tuple (g_2, g_2^x, h, h^z) is a **co-Diffie-Hellman tuple**.

When $G_1 = G_2$, these problems are the CDH and DDH problems.

A **gap co-Diffie-Hellman (co-GDH)** group pair is an ordered pair of groups (G_1, G_2) (verifying the above properties) on which the co-CDH problem is hard but the DDH problem is easy.

D.1.4 BLS signature (general case)

Let $H : \{0,1\}^* \to G_1$ be a hash function. Security of H is very important: see [11], for more information.

Here is the general case of the BLS signature:

- **Setup:** Choose a co-GDH group pair (G_1, G_2) of prime order r. Publish a generator $g_1 \in G_1$ and a generator $g_2 \in G_2$. The group pair G_1, G_2 and the generators g_1, g_2 can be used by all users.
- **Key generation:** Choose a random $x \in [\![1, r-1]\!]$. Output the public key g_2^x and the private key x.
- **Signing:** The signature of a message $m \in \{0,1\}^*$ is $\sigma = h^x$, where $h = H(m)$.
- **Verify:** Given a message-signature pair (m, σ) and the public key g_2^x, check that (g_2, g_2^x, h, σ) is a Diffie-Hellman tuple, where $h = H(m)$.

Like the BLS signature in GDH groups (D.1.2), if the signature is correctly issued, the verification passes. Furthermore, the scheme is also secure against existential forgery under a chosen-message attack in the random oracle model (see [11]).

The proof of security needs the isomorphism φ and if there is no efficiently computable isomorphism, there are cases where there is no security. But, anyway, it is also clear that the scheme is secure against universal forgery under a key only attack.

We will now focus on finding co-GDH group pairs.

D.2 Pairings

D.2.1 Definitions

According to the thesis [32], there are three different kinds of pairings: symmetric pairing, asymmetric pairing and general pairing.

Symmetric pairing enables to create GDH groups, asymmetric pairing enables to create GDH group pairs and general pairing too but without providing the isomorphism φ.

[40] Actually this requirement is often only needed in the security proofs of the pairing-based cryptosystems.

Définition D.2.1. *Let r be a number (exceptionally, r is not necessarily prime). Let $(G_1, \cdot), (G_T, \cdot)$ be cyclic groups of order r. Let $(G_2, \cdot)$ be a group where each element has an order dividing r (G_2 is not necessarily cyclic). A* ***(general bilinear) pairing*** *is an efficiently computable function*

$$e : G_1 \times G_2 \to G_T$$

such that:

1. *(Non degeneracy) $e(g_1, g_2) = 1$ for all $g_2 \in G_2$ if and only if $g_1 = 1$, and similarly, $e(g_1, g_2) = 1$ for all $g_1 \in G_1$ if and only if $g_2 = 1$.*
2. *(Bilinearity) $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$ for all $(g_1, g_2) \in G_1 \times G_2$ and for all $(a, b) \in \mathbb{Z}^2$*

Définition D.2.2. *Let r be a prime number. An* ***asymmetric (bilinear) pairing*** *is a pairing*

$$e : G_1 \times G_2 \to G_T$$

such that there is an efficiently computable isomorphism

$$\varphi : G_2 \to G_1$$

In particular G_2 is a cyclic group of order r.

Définition D.2.3. *Let r be a prime number. A* ***symmetric (bilinear) pairing*** *is a pairing*

$$e : G_1 \times G_2 \to G_T$$

where $G_1 = G_2 = G$.

A symmetric pairing is an asymmetric pairing where the isomorphism φ is the identity.

D.2.2 GDH groups and co-GDH group pairs with pairings

First let consider a symmetric pairing e over a group G. In this case, CDH is easy in G because, checking that (g, g^x, g^y, g^z) is a Diffie-Hellman tuple is equivalent ot check that:

$$e(g, g^z) = e(g^x, g^y)$$

Indeed: $e(g, g^z) = e(g, g)^z$ and $e(g^x, g^y) = e(g, g)^{xy}$. Since $e(g, g) \neq 1$ (because of the definition of a symmetric pairing), $e(g, g)$ is a generator of the cyclic group G_T and so $e(g, g)^z = e(g, g)^{xy}$ if and only if $xy = z$.

A similar proof enables to prove that if an asymmetric pairing e over groups G_1, G_2 is known, (G_1, G_2) is a co-GDH group pair.

Let consider a general pairing with r a prime number. Let g_2 be any element of $G_2 \setminus \{1\}$. g_2 generates a subgroup H in G_2. Let reduce the pairing to:

$$e : (G_1, H) \to G_T$$

The group pair (G_1, H) is quite a co-GDH pair: only the isomorphism φ may not exist.

D.2.3 Pairings over elliptic curves

In practice, pairings are often done over elliptic curves. There are several pairings over elliptic curves: Weil pairing, Tate pairing, Eta pairing, Ate pairing and Twisted Ate pairing (and maybe others).

We will focus on the Tate pairing which is often faster than the Weil pairing (and Ate or Eta pairing in some cases) and which quite a lot used.

Firstly, we need a definition:

Définition D.2.4. *Let E be an elliptic curve defined over a finite field $K = \mathbb{F}_q$. Let $G \subset E(\mathbb{F}_q$ be a cyclic group of order r. The* ***embedding degree*** *of G is the smallest positive integer such that $r | q^k - 1$.*

Notations of the definition are used for the remaining of this annex.

In practice and approximatively[41], the first input of a Tate pairing is often an element of G, the second input is often a point of the elliptic curve $E(\mathbb{F}_{q^k}$ (the curve E considered as an elliptic curve in $\mathbb{F}_{q^k}$ — with some optimizations, if $k = 2d$ is event, the second input is a point of an elliptic curve over $\mathbb{F}_q$); the result is an element of $\mathbb{F}_{q^k}$.

A point of an elliptic curve over a field $\mathbb{F}_l$ is an ordered pair (x, y) of two elements of $\mathbb{F}_l$. But for one given x, there are at most 2 corresponding y. Thus, in some cryptosystems, only the x-coordinate may be stored. This method is called **point reduction**. But, it is also possible to represent a point of an elliptic curve as x and 1 bit hich indicate which y to choose.

D.2.4 Security notions, speed and size

According to [11], the best known algorithm for solving the co-CDH problem in (G_1, G_2) is to compute the discrete log in G_1. There are two main known algorithms for that purpose:

- **MOV** ([33]): the discrete-log problem in G_1 may be transformed in a discrete log problem in $\mathbb{F}_{q^k}$.
- **Generic** (Baby-step-giant-step, Pollard's Rho or Pollard's Lambda): these algorithms have a running time proportional to $\sqrt{r}$.

Here is a summary array (computed from recommendation of document [37]):

Estimated security (bits)	q^k size (bits)	r size (bits)
80	1024	160
112	2048	224
128	3072	256

But in low-characteristic fields (when q is a power of 2 or 3, for example), it seems there are special attacks and q^k shall be greater, according to the thesis [32].

But for speed purpose, q and q^k shall be small. Furthermore in a lot of cases, the signature is a point of the elliptic curve E and its size is $q + 1$ thanks to point compression.

In the further proposed elliptic curves, $q \approx r$ and this is an optimal choice because anyway r cannot be a lot greater than q.

D.3 Practical choices of elliptic curve

The most important point is to find curves where q relatively small.

In this annex, we restrict to two kinds of curves: MNT curves and BN curves.

D.3.1 MNT curves

These curves have been discovered by Miyaji, Nakabayashi and Takano in [34] and are called type D curve in the thesis.

Their embedding degree is 6. For 80 bits security, q size shall have about 171 bits. 160 bits might be sufficient for q.

These curves are implemented in the PBC[42] software and in Miracl[43] (in particular) using an optimized version of the Tate pairing.

When no optimizations are done, it is possible to use the trace map to find an efficiently computable isomorphism from G_2 to G_1 (see the thesis [32] for more information). But the used optimizations prevent from using the trace map (because optimizations lead to use a G_2 group where each element has a trace equal to zero).

The security proof of the BLS scheme and a lot of other schemes are no more correct. But it seems these optimizations are used anyway.

[41]This is not really true. This paragraph is only here to give an idea of the size of inputs and outputs of a Tate pairing.

[42]`http://crypto.stanford.edu/pbc`

[43]`http://www.shamus.ie`

D.3.2 BN curves

BN curves have be discovered by Barreto and Naehrig in [5]. They are called type F curves. Their embedding degree is $k = 12$. On the one hand, it enable to use only 160 bits for q for 80 security bits, but in the other hand, if they are not optimized a lot, the computation might be very very long.

These curves are implemented in PBC with the Tate pairing but they are not very optimized.

Shigeo Mitsunari releases a very efficient software[44] to compute pairing using BN curves: [8]. The software provides an Ate pairing with a fixed q (of 254 bits) in less than 1 ms (which lead to a BLS signature verification in about 2 ms, which is very fast, see other benchmarks in annex E) on an "Intel Core i7 2.8GHz processor". Unfortunately it really need a 64 bits Intel/AMD microprocessor and may not be used in controller of an aggregates based system. Furthermore the used q cannot be easily changed.

[44] `http://homepage1.nifty.com/herumi/crypt/ate-pairing.html`

E Benchmarks

Performances of aggregates based system are quite important. In particular, verifying systems shall be able to verify a lot aggregates per second. For aggregating system, it is often less important, because aggregation is rarely done contrary to verification.

There are three main factors which determine the speed:

- Number of tags read by a tag interrogator per second.
- Verifying algorithm complexity.
- Cryptographic primitives (symmetric: hash functions, HMAC, AES, . . . ; and asymmetric ones: signature mainly)

For the first point, some experimental tests show that current tag interrogators can read between 5 and 50 tags per second, depending on the tag interrogator. Alien tag reader seems to be the best one.

For the second point, no real tests have been done but it seems verifying algorithms are quite fast. Tthere theoretical complexity is less than $O(n^2)$ where n is the number of read tags.

For the last point, benchmarks have been made. Two computer have been used:

- a Dell laptop "Latitude D600" (not used in verifying system — for comparison):
  - RAM: 1 Go
  - Processor: Pentium M 2.00 GHz (cache: 2Mo)
  - OS: XUbuntu 10.04 (compiler: gcc version 4.4.3, java: openjdk 1.6, gmp version 4.3.2)
- a Mini PC (Serveur Embedded 4310-JSK — used in verifying system):
  - RAM: 1 Go
  - Processor: VIA Eden Processor 500MHz
  - OS: Debian 5 (compiler: not used — benchmarks were compiled by laptop —, java: openjdk 1.6, gmp version 4.2.2)

Java is used a lot because all aggregating algorithms have been designed in Java.

There are different benchmarks:

- Java benchmarks (classical cryptography) with BouncyCastle 145 (BC)
- Java benchmarks (classical cryptography) with ViaPadlock[45] (an engine able to do some cryptographic operations, in VIA microprocessor only). Via PadLock engine also has a True Random Number Generator which uses random electrical noise.
- Java benchmarks (for BLS signatures) with JPBC (SVN version revision 197 modified to work with our computer and to produce really short signatures[46]) with PBC wrapper.
- PBC benchmarks (without JPBC) for BLS signatures (with several elliptic curves of type MNT)
- Miracl benchmarks for BLS signatures (with one elliptic curve of type MNT)

In addition, speed tests of "openssl" (openssl speed) are given for comparison, for signature. No confidence interval has been computed. Actually, it can be a good idea to create a Java wrapper to "openssl", because "openssl" is faster than Java BouncyCastle. "PBC" and "Miracl" need also a Java wrapper (because JPBC is not a direct wrapper to BLS signature of PBC and is quite slower).

Time and time errors are computed as is:

[45] `http://www.via.com.tw/en/initiatives/padlock/hardware.jsp`

[46] Before the modification, the BLS signature was a complete point of the elliptic curve. Now it is the x-coordinate, as indicated in `http://crypto.stanford.edu/pbc/manual/ch02s02.html`

- For Java benchmarks, since the timer precision ("System.GetCurrentMillis") has only a precision of 1 milliseconds, 10,000 executions of symmetric cryptographic primitives and 100 executions of asymmetric primitives (with the same key but different random messages) are timed. This timing is again executed 20 times with different random keys (but same precomputed parameters, for DSA, for instance). A 95% confidence interval is then computed over these 20 values.
- For PBC benchmarks, the used timing function is "gettimeofday". The precision is about 1 us. This is sufficient to time each call of a cryptographic primitive. Each function is executed 100 times with the same random key but different random messages. A 95% confidence interval is then computed over these 100 values. Since the same key is used for one execution, confidence intervals are not very accurate. Note: the hash function time is not counted in benchmarks.
- For Miracl benchmarks, benchmarks are the same as for PBC, except the message is always the same. Thus confidence interval are even less accurate. Note: the hash function time is not counted in benchmarks.

A lot of benchmarks have been done but here is only cryptographic functions of about 80 bits of security (except AES whose security strength is 128 bits).

Cryptographic	Pentium M 2 GHz	Via Eden 500 MHz	
function	Java + BC	Java + BC	Java + Via PadLock
SHA-1	0.003 ± 0.000 ms	0.034 ± 0.000 ms	0.012 ± 0.005 ms
HMAC-SHA-1	0.011 ± 0.000 ms	0.106 ± 0.002 ms	not implemented
AES-128-CFB — encryption	0.005 ± 0.001 ms	0.048 ± 0.000 ms	0.015 ± 0.005 ms
AES-128-CFB — decryption	0.005 ± 0.001 ms	0.052 ± 0.006 ms	0.018 ± 0.007 ms
AES-128-OFB — encryption	0.004 ± 0.000 ms	0.047 ± 0.000 ms	0.011 ± 0.000 ms
AES-128-OFB — decryption	0.004 ± 0.000 ms	0.044 ± 0.000 ms	0.018 ± 0.007 ms
- AES-128-CCM — encryption	0.015 ± 0.000 ms	0.171 ± 0.000 ms	not implemented
AES-128-CCM — decryption	0.020 ± 0.002 ms	0.179 ± 0.008 ms	not implemented
DSA[47]— signature	6.4 ± 0.2 ms	68.6 ± 0.9 ms	4.7 ± 1.3 ms
DSA[47]— verification	12.6 ± 0.2 ms	136.8 ± 0.9 ms	8.6 ± 1.4 ms
ECDSA[48]— signature	22.8 ± 0.3 ms	172.2 ± 1.5 ms	not implemented
ECDSA[48]— verification	22.2 ± 0.2 ms	223.3 ± 1.2 ms	not implemented

Table 1: Benchmark of classical cryptographic primitives (with 80 bits of security except for AES)

In summary, symmetric cryptographic primitives are not a problem, contrary to asymmetric ones whose implementation need to be carefully chosen. If short signatures are required, PBC and Miracl implementations of BLS are quite slow with mini PC (but improvements can be made, by using Via PadLock for example). There are other short signatures like the ones described in [10] which require only one pairing instead of two: this approximatively divides by 2 the verification time.

Notice also that all the benchmarked implementations are not (necessarily) secured against side-channel attacks like timing attack. If a high level of security is required, it is not recommended to use any of the previous software without deep tests.

[47]DSA (160, 1024) with SHA-1

[48]ECDSA with B-163 (FIPS elliptic curve whose curve order has 163 bits and so signature has 326 bits) and with SHA-256

[49]Names of curves are the name of "param" files of PBC (without extension ".param").

Program	Elliptic curve[49]	q size	n size	Sign/Ver.	Pentium M	Via Eden
PBC	d159	159 bits	158 bits	sign	2.1 ± 0.1 ms	15.1 ± 0.0 ms
PBC	d159	159 bits	158 bits	verify	28.8 ± 0.2 ms	243.1 ± 0.7 ms
JPBC	d159	159 bits	158 bits	sign	2.3 ± 0.1 ms	21.1 ± 0.5 ms
JPBC	d159	159 bits	158 bits	verify	34.1 ± 0.1 ms	308.6 ± 2.4 ms
PBC	d277699-175-167	175 bits	167 bits	sign	2.7 ± 0.1 ms	19.8 ± 0.1 ms
PBC	d277699-175-167	175 bits	167 bits	verify	35.5 ± 0.6 ms	309.4 ± 0.7 ms
JPBC	d277699-175-167	175 bits	167 bits	sign	2.9 ± 0.1 ms	27.6 ± 0.3 ms
JPBC	d277699-175-167	175 bits	167 bits	verify	41.1 ± 0.1 ms	388.9 ± 4.3 ms
PBC	d201	201 bits	180 bits	sign	3.1 ± 0.1 ms	25.3 ± 0.5 ms
PBC	d201	201 bits	180 bits	verify	44.4 ± 0.5 ms	398.2 ± 0.9 ms
JPBC	d201	201 bits	180 bits	sign	3.6 ± 0.1 ms	36.5 ± 3.4 ms
JPBC	d201	201 bits	180 bits	verify	50.8 ± 0.8 ms	490.3 ± 6.6 ms
Miracl		159 bits	158 bits	sign	1.9 ± 0.3 ms	13.6 ± 0.1 ms
Miracl		159 bits	158 bits	verify	26.1 ± 1.2 ms	176.0 ± 0.6 ms

Table 2: Benchmark of BLS signature over MNT curves (q is the order of the used finite field and n is the order of the considered cyclic group of the elliptic curve — for 80 bits of security, it is recommended that q size is more than 160 bits and n is more than $\frac{1024}{9} \approx 170.7$, because 6 is the embedded degree of MNT curves)

F Tag features

This annex gives a non-exhaustive list of current HF and UHF tags. Only publicly available information is given and they may have some errors. Interrogation points "?" indicates the detail is unknown or has been guessed from other data given by the manufacturer and might be not true.

A summary of this list can be found in section 7.1.4.

Here are website of the manufacturers:

- ST: `http://www.st.com/stonline/products/families/memories/rfid/rfid.htm`
- Legic: `http://www.legic.com`
- Mifare: `http://www.mifare.net`
- Infineon: `http://www.infineon.com`
- Ask: `http://www.ask-rfid.com`
- Alien technology: `http://www.alientechnology.com`
- Impinj (Monza 4): `http://www.impinj.com/products/monza4tagchips.aspx`
- SecureRF: `http://www.securerf.com/sensorrfidtags.shtml`

The Mifare SmartMX is a programmable chip: a lot of features can be implemented (as soon as they can be implemented using the microprocessor and the cryptographic co-processor of the chip). That is why some details of this chip are "n/a": they depend a lot of the software of the chip. Infineon SLE 66CLxxxPE and SLE 78CLxxxPE seem (but it is not sure) to be programmable too.

Product				
Manufacturer	Product	Standard	Read distance	Unique Id[a]
ST	Short range	ISO 14443-B	10-15 cm	64 bits
	Long range	ISO 15693	1.5 m	64 bits
Legic	Advant	ISO 15693	70 cm	?
Mifare (NXP)	Plus	ISO 14443-A	10 cm ?	32-56 bits
	Ultralight	ISO 14443-A	10 cm	56 bits
	Ultralight C	ISO 14443-A	10 cm	56 bits
	DESFire EV1	ISO 14443-A	10 cm ?	56 bits
	SmartMX	ISO 14443-A	10 cm ?	yes
Infineon	SRF 55VxxP (my-d vicinity)	ISO 18000-3 mode 1	1.5 m	yes
	SRF 55VxxS (my-d vicinity secure)	ISO 18000-3 mode 1	1.5 m	yes
	SRF 66Vxxxx (PJM *)	ISO 18000-3 mode 2	1.5 m	yes
	SLE 66RxxP (my-d NFC)	ISO 14443-A	10 cm	32 bits
	SLE 66RxxS (my-d proximity)	ISO 14443-A	10 cm	32 bits
	SLE 66CLxxxPE	ISO 14443-A/B + ISO 18092 passive	10 cm	?
	SLE 78CLxxxPE	ISO 14443-A/B + ISO 18092 passive	10 cm	?
Ask	CTM-512-B (C.ticket)	ISO 14443-B	10 cm	64 bits
	Other C.tickets	ISO 14443-A/B or ISO 18092	10 cm	32-64 bits
Alien	Higgs 3	C1G2 + custom commands		64 bits
Impinj	Monza 4	C1G2 + custom commands		48 bits
SecureRF	Lime	C1G2 + custom commands		yes
Ask	C.Label	C1G2		64 bits

Table 3: Tag features (part 1/4)

[a]Here unique id is just an unique factory programmed serial number. It is not a unique id as defined in section 6.2.

Product			Lock[a]		
Manufacturer	Product	R/W memory	Write	Read	Kill
ST	Short range	512 – 4K bits	no	block (perma)	no
	Long range	0 – 2K bits	no	no	LRi2K only
Legic	Advant	?	?	?	?
Mifare (NXP)	Plus	16 – 32 Kbits	block (key)	block (key)	no
	Ultralight	384 bits + 32 bits OTP	no	yes	no
	Ultralight C	1536 bits	maybe	maybe	no
	DESFire EV1	16 - 64 Kbits	yes	yes	no
	SmartMX	see website	n/a	n/a	n/a
Infineon	SRF 55VxxP	2.5 – 10 Kbits	no (?)	no (?)	no
	SRF 55VxxS	2.5 – 10 Kbits	yes (?)	yes (?)	no
	SRF 66Vxxxx	1 – 10 Kbits	yes ? (password)	yes (password)	no
	SLE 66RxxP	4 – 32 Kbits	block ?	block	no
	SLE 66RxxS	4 – 32 Kbits	block ?	block	no
	SLE 66CLxxxPE	see website	n/a	n/a	n/a
	SLE 78CLxxxPE	see website	n/a	n/a	n/a
Ask	CTM-512-B	512 bits	block	block	?
	Other C.tickets	512 – 8K bits	yes	yes	?
Alien	Higgs 3	512 + 96 bits	block (password)	block (password)	yes
Impinj	Monza 4	up to 512 + 128 bits	no	block (password)	yes
SecureRF	Lime	?	?	?	yes
Ask	C.Label	64 to 512 bits	?	?	yes

Table 4: Tag features (part 2/4)

[a]Locks can be performed on the whole memory (“yes”) or per block (“block”). The lock can be permanent (“perma” —impossible to unlock) or can require a password (“password”) or a key (“key”) for a more complex authentication mechanism.

Product			
Manufacturer	Product	Unclonable[a]	Cryptographic primitives[b]
ST	Short range	unique id[c]	no
	Long range	no	no
Legic	Advant	yes	DES, 3DES and sometimes AES
Mifare (NXP)	Plus	yes	AES, RNG
	Ultralight	no	no (I suppose)
	Ultralight C	yes	3DES, no RNG
	DESFire EV1	yes	DES, 3DES, 3KDES(?), AES, RNG
	SmartMX	yes	DES, 3DES, AES, RSA, ECC, RNG
Infineon	SRF 55VxxP	no	no
	SRF 55VxxS	yes	?
	SRF 66Vxxxx	no	no
	SLE 66RxxP	no	no
	SLE 66RxxS	yes	?
	SLE 66CLxxxPE	yes	3DES, RSA (2048 bits), ECC (521 bits), True RNG
	SLE 78CLxxxPE	yes	AES, 3DES, RSA (4096 bits), ECC (521 bits), True RNG
Ask	CTM-512-B	yes	3DES
	Other C.tickets	yes[d]	3DES[d]
Alien	Higgs 3	unique id ?[e]	?
Impinj	Monza 4	no	no
SecureRF	Lime	yes	Algebraic Eraser^{TM}
Ask	C.Label	no	no

Table 5: Tag features (part 3/4)

[a]Unclonable features are one of the solutions of section 6.2. In this list, there is only two possibilities: randomized encryption ("yes") and unique id with zero-knowledge proof ("unique id"). Password authentication may provide a lightweight unclonability feature. Password support is indicated in authentication column of the next table.

[b]See section 7.1.4 (more precisely the end of the paragraph "HF tags").

[c]SriX4K only

[d]Only for some of them.

[e]Depends a lot on the (unknown) Alien dynamic authentication.

Product			
Manufacturer	Product	Authentication	Comments
ST	Short range	no	
	Long range	no	
Legic	Advant	unknown	
Mifare (NXP)	Plus	symmetric (AES - per sector[a])	
	Ultralight	no	
	Ultralight C	symmetric (3DES)	
	DESFire EV1	symmetric (AES) ?	
	SmartMX	n/a	programmable
Infineon	SRF 55VxxP	?	
	SRF 55VxxS	unknown (64 bits) ?	
	SRF 66Vxxxx	48 bits password + ?	
	SLE 66RxxP	?	
	SLE 66RxxS	unknown (64 bits) ?	
	SLE 66CLxxxPE	?	programmable ? + side-channel attacks countermeasures
	SLE 78CLxxxPE	?	programmable ? + side-channel attacks countermeasures
Ask	CTM-512-B	symmetric (3DES)	
	Other C.tickets	symmetric (3DES)[b]	
Alien	Higgs 3	pass[c] + Alien dynamic authentication[d]	
Impinj	Monza 4	pass[c]	QT™[e] technology support
SecureRF	Lime	pass[c] + asymmetric (Algebraic Eraser™)	current versions are battery assisted
Ask	C.Label	pass[c]	

Table 6: Tag features (part 4/4)

[a] Memory is divided in sectors. A different key can be used for each sector.
[b] Only for some of them
[c] Password. C1G2 tags shall have a kill password of 32 bits and may have an access password.
[d] Symmetric authentication of the Higgs 3 chip. The algorithm is proprietary.
[e] See `http://www.impinj.com/products/qt-technology.aspx` for further information.

Contents

References

[1] Alien Technology®. *Alien Technology® Bolsters Higgs-3^{TM} RFID IC Security with "Dynamic Authentication"*. Sept. 2009. URL: `http://www.alientechnology.com/newsevents/2009/press091609.php`.

[2] Alien Technology®. *Pharmaceutical Shifts Towards UHF RFID for Savings.* White paper, Alien. URL: `http://www.alientechnology.com/docs/WP_UHF_RFIDPharmaceutical.pdf`.

[3] I. Anshel et al. "Key agreement, the Algebraic Eraser^{TM}, and lightweight cryptography". In: *Algebraic methods in cryptography: AMS/DMV Joint International Meeting, June 16-19, 2005, Mainz, Germany: International Workshop on Algebraic Methods in Cryptography, November 17-18, 2005, Bochum, Germany.* Vol. 418. American Mathematical Society. 2006, p. 1.

[4] G. Avoine and P. Oechslin. "RFID traceability: A multilayer problem". In: *Financial Cryptography and Data Security* (2005), pp. 125–140.

[5] P. Barreto and M. Naehrig. "Pairing-friendly elliptic curves of prime order". In: (2006), pp. 319–331.

[6] L. Batina et al. "An elliptic curve processor suitable for RFID-tags". In: *Int. Assoc. for Cryptologic Research ePrint Archive* (2006).

[7] B. Belcher, M. El-Said, and G. Nezlek. "Lightweight RFID authentication protocol: An experimental study". In: *30th International Conference on Information Technology Interfaces, 2008. ITI 2008.* 2008, pp. 583–588.

[8] Jean-Luc Beuchat et al. *High-Speed Software Implementation of the Optimal Ate Pairing over Barreto-Naehrig Curves.* Cryptology ePrint Archive, Report 2010/354. `http://eprint.iacr.org/`. 2010.

[9] Leonid Bolotnyy and Gabriel Robins. "Physically Unclonable Function-Based Security and Privacy in RFID Systems". In: *Pervasive Computing and Communications, 2007. PerCom '07. Fifth Annual IEEE International Conference on.* 2007, pp. 211 –220. DOI: `10.1109/PERCOM.2007.26`.

[10] D. Boneh and X. Boyen. "Short signatures without random oracles and the SDH assumption in bilinear groups". In: *Journal of Cryptology* 21.2 (2008), pp. 149–177.

[11] D. Boneh, B. Lynn, and H. Shacham. "Short Signatures from the Weil Pairing". In: Springer, 2001, pp. 514–532.

[12] P. Bose et al. "On the false-positive rate of Bloom filters". In: *Information Processing Letters* 108.4 (2008), pp. 210–213.

[13] M. Braun, E. Hess, and B. Meyer. "Using elliptic curves on rfid tags". In: *IJCSNS* 8.2 (2008), p. 1.

[14] N. Courtois, M. Finiasz, and N. Sendrier. *How to achieve a McEliece-based Digital Signature Scheme.* Cryptology ePrint Archive, Report 2001/010. 2001. URL: `http://eprint.iacr.org/`.

[15] Nicolas Courtois. *McEliece signature.* 2001. URL: `http://www.cryptosystem.net/mceliece`.

[16] Nicolas Courtois. *QUARTZ signature.* 2003. URL: `http://www.cryptosystem.net/quartz`.

[17] Nicolas T. Courtois. *The Dark Side of Security by Obscurity and Cloning MiFare Classic Rail and Building Passes Anywhere, Anytime.* Cryptology ePrint Archive, Report 2009/137. 2009. URL: `http://eprint.iacr.org/`.

[18] Q. Dang. "Recommendation for Applications Using Approved Hash Algorithms". In: *NIST Special Publication* 800 (2009), p. 107.

[19] Srini Devadas. *Physical Unclonable Functions and Applications.* URL: `http://people.csail.mit.edu/rudolph/Teaching/Lectures/Security/Lecture-Security-PUFs-2.pdf`.

[20] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2.* RFC 5246 (Proposed Standard). Updated by RFCs 5746, 5878. Internet Engineering Task Force, Aug. 2008. URL: `http://www.ietf.org/rfc/rfc5246.txt`.

[21] N. Draft. "Special Publication 800-38C". In: *Recom mendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and confidentiality, US Doc/NIST* (2004).

[22] V. Dubois et al. "Practical cryptanalysis of SFLASH". In: *Proceedings of the 27th annual international cryptology conference on Advances in cryptology.* Springer-Verlag. 2007, pp. 1–12.

[23] M. Dworkin. "NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC". In: *US National Institute of Standards and Technology, November* (2007).

[24] EPCGlobal$^{\text{TM}}$. *EPC Tag Data Standard (TDS) v. 1.4.* June 2008. URL: `http://www.epcglobalinc.org/standards/tds/tds_1_4-standard-20080611.pdf`.

[25] EPCGlobal$^{\text{TM}}$. *EPCGlobal^{TM} Tag Class Structure.* Nov. 2007. URL: `http://www.epcglobalinc.org/standards/TagClassDefinitions_1_0-whitepaper-20071101.pdf`.

[26] EPCGlobal$^{\text{TM}}$. *UHF Class 1 Gen 2 Standard v. 1.2.* May 2008. URL: `http://www.epcglobalinc.org/standards/uhfc1g2`.

[27] Hanser. *Metalcraft introduces destructible option for RFID Windshield Tag.* URL: `http://www.hanser.com/2009/news/metalcraft-introduces-destructible-option-for-rfid-windshield-tag`.

[28] J. Jonsson and B. Kaliski. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1.* RFC 3447 (Informational). Internet Engineering Task Force, Feb. 2003. URL: `http://www.ietf.org/rfc/rfc3447.txt`.

[29] A. Kalka, M. Teicher, and B. Tsaban. "Cryptanalysis of the Algebraic Eraser and short expressions of permutations as products". In: *Arxiv preprint arXiv:0804.0629* (2008).

[30] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication.* RFC 2104 (Informational). Internet Engineering Task Force, Feb. 1997. URL: `http://www.ietf.org/rfc/rfc2104.txt`.

[31] J.C. Laprie. *Dependability: basic concepts and terminology in English, French, German, Italian, and Japanese.* Dependable Computing and Fault-Tolerant Systems. Springer-Verlag Wien New York, 1991.

[32] B. Lynn. "On the implementation of pairing-based cryptosystems". PhD thesis. Citeseer, 2007.

[33] A.J. Menezes, T. Okamoto, and S.A. Vanstone. "Reducing elliptic curve logarithms to logarithms in a finite field". In: *IEEE Transactions on Information Theory* 39.5 (1993), pp. 1639–1646.

[34] A. Miyaji, M. Nakabayashi, and S. Takano. "New explicit conditions of elliptic curve traces for FR-reduction". In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 84.5 (2001), pp. 1234–1243.

[35] A.D. Myasnikov and A. Ushakov. "Cryptanalysis of the Anshel-Anshel-Goldfeld-Lemieux key agreement protocol". In: *Groups-Complexity-Cryptology* 1.1 (2009), pp. 63–75.

[36] NIST. *FIPS 186-3, Digital Signature Standard (DSS).* 2009. URL: `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`.

[37] NIST. *Recommendation for Key Management.* Mar. 2007. URL: `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf`.

[38] R. Nithyanand, G. Tsudik, and E. Uzun. *Readers behaving badly: Reader revocation in pki-based rfid systems.* Tech. rep. Cryptology ePrint Archive, Report 2009/465, 2009.

[39] T. Okamoto. "Provably secure and practical identification schemes and corresponding signature schemes". In: *Advances in Cryptology—CRYPTO'92.* Springer. 1993, pp. 31–53.

[40] J. Patarin, N. Courtois, and L. Goubin. "QUARTZ, 128-bit long digital signatures". In: *Topics in Cryptology—CT-RSA 2001* (2001), pp. 282–297.

[41] RFC. *RFC 4253 - The Secure Shell (SSH) Transport Layer Protocol.* URL: `http://www.ietf.org/rfc/rfc4253.txt`.

[42] C.P. Schnorr. "Efficient signature generation by smart cards". In: *Journal of cryptology* 4.3 (1991), pp. 161–174.

[43] P. Tuyls and L. Batina. "RFID-tags for Anti-Counterfeiting". In: *Topics in Cryptology–CT-RSA 2006* (2006), pp. 115–131.

[44] Verayo. *Verayo PUF RFID.* URL: `http://www.verayo.com/product/pufrfid.html`.

[45] Wikipedia. *Birthday problem.* Aug. 2010. URL: http://en.wikipedia.org/wiki/Birthday_problem.

[46] Wikipedia. *Bloom Filter.* June 2010. URL: http://en.wikipedia.org/wiki/Bloom_filter.

[47] Wikipedia. *Diffie–Hellman key exchange.* Aug. 2010. URL: http://en.wikipedia.org/wiki/Diffie\%E2\%80\%93Hellman_key_exchange.

[48] Wikipedia. *ID-based cryptography.* Aug. 2010. URL: http://en.wikipedia.org/wiki/ID-based_cryptography.

[49] Wikipedia. *ID-based encryption.* Aug. 2010. URL: http://en.wikipedia.org/wiki/ID-based_encryption.

[50] Wikipedia. *Recommendation for Block Cipher Modes of Operation.* Aug. 2010. URL: http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

[51] Wikipedia. *Side-channel attack.* Aug. 2010. URL: http://en.wikipedia.org/wiki/Side-channel_attack.

[52] Wikipedia. *Stream cipher.* Aug. 2010. URL: http://en.wikipedia.org/wiki/Stream_cipher.

[53] www.keylength.com. *Key length recommendation.* Aug. 2010. URL: http://www.keylength.com/en/4.

[54] www.sciengines.com. *Break DES in less than a single day.* URL: http://www.sciengines.com/joomla/index.php?option=com_content&view=article&id=99:des-in-1-day&catid=35:newsannounce&Itemid=58.

[55] www.telecom.gouv.fr. *RFID frequencies.* URL: http://www.telecom.gouv.fr/rubriques-menu/organisation-du-secteur/dossiers-sectoriels/etiquettes-electroniques-rfid/les-frequences-453.html.

[56] F. Zhang and K. Kim. “Signature-masked authentication using the bilinear pairings”. In: *Cryptology and Information Security Laboratory (CAIS), Information and Communications University, Tech. Rep* (2002).

[57] Y. Zheng. “Digital signcryption or how to achieve cost (signature & encryption) << cost (signature) + cost (encryption)”. In: *Advances in Cryptology-CRYPTO'97: 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 1997. Proceedings.* Springer. 1997, p. 165.

[58] Yuliang Zheng. *Signcryption Central.* URL: http://www.signcryption.org.