ACTION: Breaking the Privacy Barrier for RFID Systems

Li Lu, Jinsong Han, Renyi Xiao, and Yunhao Liu

Department of Computer Science & Engineering, Hong Kong University of Science & Technology, Hong Kong, China

***Abstract*—In order to protect privacy, Radio Frequency Identification (RFID) systems employ Privacy-Preserving Authentication (PPA) to allow valid readers to explicitly authenticate their dominated tags without leaking private information. Typically, an RF tag sends an encrypted message to the reader, then the reader searches for the key that can decrypt the cipher to identify the tag. Due to the large-scale deployment of today's RFID systems, the key search scheme for any PPA requires a short response time. Previous designs construct balance-tree based key management structures to accelerate the search speed to $O(\log N)$, where N is the number of tags. Being efficient, such approaches are vulnerable to compromising attacks. By capturing a small number of tags, compromising attackers are able to identify other tags that have not been corrupted. To address this issue, we propose an Anti-Compromising authenticaTION protocol, ACTION, which employs a novel sparse tree architecture, such that the key of every tag is independent from one another. The advantages of this design include: 1) resilience to the compromising attack, 2) reduction of key storage for tags from $O(\log N)$ to $O(1)$, which is significant for resource critical tag devices, and 3) high search efficiency, which is $O(\log N)$, as good as the best in the previous designs.**

Keywords-RFID; privacy; authentication; compromising

I. INTRODUCTION

Due to the low cost and easy deployment, Radio-Frequency Identification (RFID) has been an important enabling technology for everyday applications, such as retailing, medical-patient management, access control [1], logistics and supply chain management [2, 3]. In RFID systems, RF tags emit their unique serial numbers to RF readers. Without privacy protection, however, any reader can identify a tag ID via the emitted serial number. Indeed, a malicious reader can easily perform bogus authentications with detected tags to retrieve sensitive information within its scanning range. Currently, many companies embed RF tags into items. As these tags contain unique information about the items, a customer carrying the tags is subject to silent tracking from unauthorized readers. Sensitive personal information might be also leaked: details about an illness inferred by the purchase of certain pharmaceutical products; the malls she shops at; the types of items she prefers to buy, and so on. Clearly, a secure RFID system must meet two requirements. First, valid readers must be able to identify valid tags. Second, misbehaving readers should not be able to retrieve private information from valid tags.

In order to protect privacy, Privacy-Preserving Authentication (PPA) is introduced into the interactive procedure between RFID readers and tags [4]. To achieve PPA, an RFID tag performs a cryptography enabled challenging-response procedure with a reader [5]. For example, we can let each tag share a key with the reader. During authentication, the reader first probes a tag via a query message with a nonce. Instead of answering the query in plaintext, the tag replies to the reader with the encrypted nonce. The reader searches all the keys that it holds in the back-end database. If the tag is valid, the reader can find a proper key to recover the authentication message, and thereby identify the tag. (For simplicity, we use the term "reader" to denote the reader device as well as the back-end database in the following). Using PPA, any invalid tag will not be accepted since it cannot provide a proper cipher related to a key owned by the reader. Meanwhile, the valid tag does not expose its identity to any third party in PPAs since only valid readers know the key used for encrypting messages. A malicious reader cannot identify a user via probing the valid tag.

Although it is simple and secure, such a PPA based design suffers poor scalability. Upon receiving a nonce cipher, the reader needs a prompt lookup to locate a key in the database. Clearly, the search complexity is $O(N)$, where N is the number of all the possible tags, even only a small portion of them are in the reader's range. In today's large-scale RFID systems, N is often as large as hundreds of millions, and thousands of tags may respond to a reader simultaneously, demanding a fast key-search method as well as a carefully designed key-storage structure. Hence, balance-tree based schemes [6-9] employ key-sharing infrastructure to accelerate the authentication procedure, in which the lookup complexity is $O(\log N)$.

The balance-tree based approaches are efficient, nevertheless, not secure due to the key-sharing infrastructure. As the infrastructure used by those approaches is static, each tag, more or less, shares some common keys with other tags (in this paper, we use *normal* tags to denote tags that are not tampered with). Consequently, compromising one tag might reveal information of other tags [6, 9]. L. Lu, *et al* evaluate the damage caused by the compromising attack to balance-tree based approaches [9]. By compromising only twenty tags, an adversary can achieve a nearly 100% probability of successfully track normal tags in a balance-tree based RFID system containing 2^{20} tags [10].

L. Lu, *et al* propose a dynamic key-updating scheme [9], SPA, for balance-tree base approaches to mitigate the impact of compromising attacks. SPA reduces the number of keys shared among compromised and normal tags, and alleviates

the damage caused by compromising attacks. SPA, however, does not eliminate the impact of compromising attacks. For instance, using SPA in an RFID system with 2^{20} tags, the probability of tracking normal tags is close to 60% after an adversary compromises twenty tags [9].

Another drawback for balance-tree based PPAs is the large space needed for storing keys in each tag. Balance-tree based approaches require each tag to hold $O(\log_\delta N)$ keys, and the reader to store $\delta \times N$ keys, where δ is a branch factor of the key tree. Due to the limited memory capacity of RF tags, existing PPAs are difficult to apply in current RFID systems.

In order to address the above issues, we propose an Anti-Compromising authenticaTION protocol, called ACTION. By employing a sparse tree to organize keys, ACTION generates completely independent keys for tags, so that compromised tags have no keys that correlate with the normal ones. As a result, ACTION can effectively defend against compromising attacks. We show that if an attacker can track a normal tag with a probability larger than α, it must tamper with more than $N - 1/\alpha$ tags, while in previous balance-tree based approaches, by compromising $O(\log N)$ tags, an attacker can track a normal tag with a probability close to 100% [10]. Another salient feature of this design is the low storage requirement for tags. ACTION only allows each tag to store two keys and the reader to store $O(N)$ keys, achieving high storage efficiency for both readers and tags, making this design practical for today's RF tags. We also show that ACTION retains high search efficiency in the sense that the lookup complexity is still $O(\log N)$, as good as the best of previous designs.

The rest of this paper is organized as follows. We discuss the related work in Section II. We present the ACTION protocol in Section III. In Section IV, we discuss the storage and search efficiency of ACTION. We present the security analysis in Section V, and conclude the work in Section VI.

II. Related Work

The fundamental principle of PPAs is based on HashLock [5], in which every tag shares a unique key with the reader. A tag and the reader use a challenging-response scheme to conduct authentication. Recent studies [13] show that HashLock is a secure PPA. The main drawback of HashLock is that the key search is linear to the number of tags in the system, which limits the usage of HashLock in large-scale RFID systems. Subsequent approaches in the literature are mostly aimed at improving the efficiency of key search. Juels [14] classifies those approaches into three categories.

Synchronization approaches: Such approaches [15-18] use an incremental counter to record the state of authentication. When an authentication is successfully performed, the tag increases the counter by one. The reader compares the value of a tag's counter with the record in the database. If the difference of the two counter values is in a proper window, the tag is viewed as valid and the reader synchronizes the counter record of the tag. Synchronization schemes are subject to the *desynchronization* attack [14], in which a malicious reader interrogates a tag many times such that the counter of the tag exceeds the range of the window and the reader fails to recognize a valid tag.

Time-space tradeoff approaches: OSK [15] and AO [19] employ Hellman tables [20] to improve the key-efficiency. Hellman studies the problem of breaking symmetric keys and shows that an adversary can pre-compute a Hellman table of storage size $O(N^{2/3})$, in which the adversary can search a key with the complexity of $O(N^{2/3})$. That means the key-searching efficiency of OSK or AO is also $O(N^{2/3})$. Those approaches are not sufficiently efficient for supporting large-scale RFID systems.

Balance-tree based approaches: Balance-tree based approaches [6-9] improve the key search efficiency from linear complexity to logarithmic complexity. They employ a balance-tree to organize and store keys for tags. In a balance-tree, each node stores a unique key. Keys in the path from the root to a leaf node are distributed to a tag. Each tag uses these multiple keys to encrypt the identification message. Upon receiving an encrypted message, the reader performs a Depth-First Search on the key tree with a logarithmic complexity of the system size. The balance-tree based approaches, however, are subject to the *compromising* attack [6, 9]. In a balance-tree, tags always share keys with others. Hence, hacking one tag may reveal several keys used by other tags. For example, in a binary balance-tree based RFID system containing 2^{20} tags, an adversary can identify any tag with the probability of about 90% by tampering with only twenty tags [9, 10]. To address a compromising attack, L. Lu, *et al* propose a dynamic key-updating scheme, SPA [9], for enhancing balance-tree based approaches. In SPA, the reader dynamically and recursively updates keys in the key tree and coordinates the keys with the tag after a successful identification. The key-updating scheme reduces the probability of locating a tag via compromising attacks. However, the threat from compromising attacks has not been completely relieved. For instance, in a SPA system containing 2^{20} tags, an adversary can still recognize any normal tag with a high probability (about 60%) after it tampers with twenty tags [9].

III. ACTION Design

In this section, we first discuss the motivation of this work, and then present the details of the ACTION protocol.

A. Motivation

In previous balance-tree based approaches, initially, a reader organizes a hierarchical balance-tree with a depth of $\log_\delta N$ (δ is branching factor), in which each node is assigned a unique key. The reader then monogamously maps N leaf nodes to N tags. Figure 1 plots a balance-tree for eight tags. For each tag, there is a unique shortest path from the root to the corresponding leaf node. For example, in Fig. 1, the tag T_3 obtains $k_{1,1}$, $k_{2,2}$, and $k_{3,3}$. During authentication, upon receiving a request with a nonce r from the reader, T_3 encrypts r in the way $\{k_{1,1}\{r\}, k_{2,2}\{r\}, k_{3,3}\{r\}\}$ and sends the ciphers to the reader. Upon receiving the response from T_3, the reader searches proper keys in the key tree to recover r. This is equal to exploring a path from the root to the leaf node of T_3 in the tree. At the end of identification, if such a path exists, R

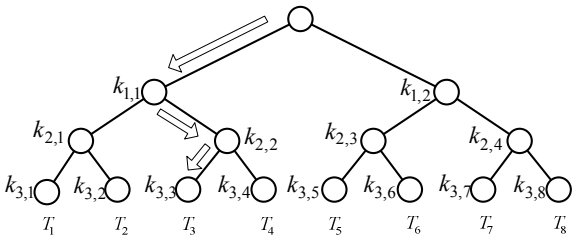

Figure 1. An example of key organization in balance-tree based PPAs.

regards T_3 as a valid tag. Clearly, the search complexity is $O(\log N)$.

The fundamental nature of balance-tree based PPAs is that a tag shares some non-leaf nodes, more or less, with other tags in the key tree. This is a fatal flaw when balance-tree based PPAs are under the compromising attack. For example, in Fig. 1, we can see that a common key, $k_{1,1}$, is shared by tags T_1, T_2, T_3, and T_4, and $k_{2,2}$ is shared by T_3 and T_4. If an adversary compromises T_3 and reveals the keys stored in T_3, the keys $k_{1,1}$ and $k_{2,2}$ are also exposed. As a result, even though T_4 is not cracked, the attacker can easily distinguish T_4 via $k_{1,1}$ and $k_{2,2}$. Even worse, the adversary can actually distinguish each normal tag by only compromising a small fraction of all tags.

Recently, L. Lu, *et al* propose a dynamic key-updating method, SPA [9], which mitigates the impact of the compromising attack. SPA updates a tag's keys from the leaf node to the root in the key tree. Each non-leaf node uses a number of state bits to record the key-updating status of its children. The 'old' keys that are used by other tags will be stored into temporary caches. The non-leaf node automatically updates its own keys if all its children have updated their keys. Compared to non-key-updating approaches, SPA is more secure because it reveals fewer keys shared between a compromised tag and normal tags to adversaries.

Using SPA, however, the probability that compromising attacks succeed [9] is still large, more than 50% in general cases. That is, for an RFID system containing 2^{20} tags, an adversary only needs to compromise 20 tags before it is able to distinguish any tag from others with a probability larger than 50%. The main reason is that the dependence among the keys of different tags is remaining. SPA reduces the number of keys correlating to normal tags, but the tags always share keys in balanced tree structure. Hence, tags are still threatened by compromising attacks.

Based on the above analysis, it is clear that the only solution to compromising attacks is to eliminate the correlation among the keys of different tags. Therefore, in this design, we intend to remove all correlations among the keys. The difficulty is that we cannot sacrifice the search efficiency as well as the storage efficiency.

B. Overview

ACTION has four components: *system initialization*, *tag identification*, *key-updating*, and *system maintenance*. In the first component, instead of using a balance-tree, we employ a sparse tree to organize keys for tags. We generate two random keys (128 bits) for each tag, denoted as leaf key k^l, and path key k^p, respectively. The k^p is corresponding to a path in the sparse tree according to its value. Each tag is associated with a leaf node in the tree after the key initialization. The leaf node thereby holds the key k^l assigned to the tag, and the path from the root to the leaf node indicates the key k^p. Since the two keys are randomly generated, keys among different tags are independent. In the second component, the reader performs a logarithmic search to identify a tag. In the third component, ACTION performs a key-updating procedure, in which ACTION employs a cryptographic hash function, such as MD5, SHA-1, to update the old key in a tag. Note that the new key is still random and independent of the keys used by other tags. ACTION also reduces the maintenance overheads in highly dynamic systems where tags join or leave frequently by using the fourth component.

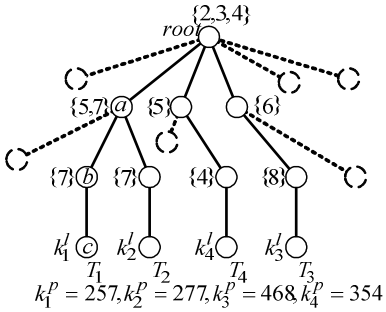

Figure 2. A key tree with four tags ($N = 4$).

C. System Initialization

We assume that there are N tags T_i, $1 \le i \le N$, and a reader R in the RFID system. We denote the sparse tree used in ACTION as S. Let δ denote the branching factor of the key tree and d denote the depth of the tree. Each tag is associated with a leaf node in S. The secret keys shared by tag T_i and reader R are denoted as k_i^p and k_i^l. Let n be the length of k_i^p and k_i^l, i.e. $|k_i^p| = |k_i^l| = n$. We split k_i^p into d parts as $k_i^p[0]||k_i^p[1]||\ldots||k_i^p[d\text{-}1]$ ('||' denotes concatenation), and the length of each $k_i^p[m]$ is n/d, $0 \le m \le d-1$. We set the branching factor, δ, of each non-leaf node in S as $2^{n/d}$, namely $d \times \log\delta = n$. For example, if we set the key length as 128 bits and $d = 32$, the branching factor of the S is $\delta = 2^{128/32} = 2^4 = 16$. In other words, each non-leaf node is able to accommodate 16 child positions in S. If the c-th child node exists in a child position of a non-leaf node j, we set c as the *index number* of this child and record c in j. Note that a non-leaf node *only* stores the index numbers for existing children.

For simplicity, we denote the set of j's index numbers as IS_j, and the element number of IS_j as IN_j, that is, $IN_j = |IS_j|$. We show an example in Fig. 2, in which the branching factor δ is 2^4. Each non-leaf node has sixteen child positions. For a non-leaf node a, as shown in Fig. 2, the reader maintains its' index number set as $IS_a = \{5, 7\}$, and the IN_a=2.

Initially, the tree is empty. Reader R generates two keys k_i^p and k_i^l uniformly at random for every tag T_i. Meanwhile, the reader divides each k_i^p into d parts, $k_i^p[0]||k_i^p[1]||\ldots||k_i^p[d\text{-}1]$, where d is the depth of key tree S. The reader distributes k_i^p and k_i^l to tag T_i and organizes k_i^p into S as follows. From the root, the reader generates a non-leaf node at each level m

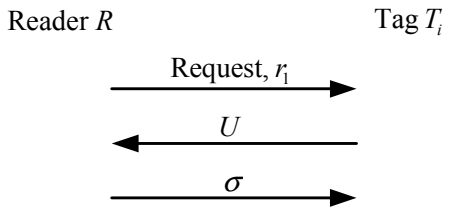

Figure 3. The authentication procedure of ACTION.

according to the corresponding $k_i^p[m]$. Specifically, after generating a node a at the level m-1 according to the $k_i^p[m\text{-}1]$, the reader will generate the $k_i^p[m]$-th child of node a, and set an index number of a as $k_i^p[m]$. For the example shown in Fig. 2, the branching factor δ of S is 16, and there are four tags in the system, denoted as T_1, T_2, T_3, and T_4. Assume that the length of path key is twelve bits. Each path key is divided into three parts, and the length of each part is four bits (because $\delta = 16$, the length of each part of a key should be $\log_2 16 = 4$ bits). The reader generates four path keys as 257, 277, 468, and 354 for tags T_1-T_4, respectively. The reader also generates four leaf keys as k_1^l, k_2^l, k_3^l, and k_4^l for T_1-T_4, respectively. For T_1, k_1^p = 257 (0010||0101||0111), thus, $k_1^p[0] = 2$, $k_1^p[1] = 5$, and $k_1^p[2] = 7$. The reader first generates a child at the root, and sets an index number as 2 ($k_1^p[0] = 2$). Here the index number 2 means the root has a child marked as node a in its second position, as illustrated in Fig. 2. Then the reader generates a child b of node a, and sets an index number of a as 5 ($k_1^p[1] = 5$). Finally, the reader generates a child c of node b, which is a leaf node c, and sets an index number of b as 7 ($k_1^p[2] = 7$). Indeed, the key organization is analogous to generating a path in tree S. In the above example, the path of T_1 is $root \rightarrow a \rightarrow b \rightarrow c$. After the same procedures on tags T_2, T_3, and T_4, we obtain a sparse tree as illustrated in Fig. 2. The procedure is described in Algorithm 1 TagJoin.

Algorithm 1: TagJoin (Tag T, Key Tree S)

```
1:  k^p, k^l ← KeyGeneration(T);
2:  (k^p[0],…, k^p[d-1]) ← KeyDivision(k^p);
3:  Node ← GetRoot(S);
4:  for i = 0 to d – 1
5:        Add k^p[i]into Node's Index Set IS;
6:        if the k^p[i]-th child does not exist
7:              Create the k^p[i]-th child;
8:              Node ← the k^p[i]-th child;
9:        else Node ← the k^p[i]-th child;
```

D. Tag Identification

ACTION employs cryptographic hash functions to generate authentication messages and update keys. Let h denote a cryptographic hash function: $h:\{0,1\}^* \rightarrow \{0,1\}^n$, where n denotes the length of the hash value. Let N be the number of all tags in the system. The basic authentication procedure between the reader and a tag T_i ($1 \le i \le N$) includes three phases, as illustrated in Fig. 3. In the first phase, the reader R sends a "Request" with a random number r_1 (a nonce) to tag T_i. In the second phase, upon receiving "Request", tag T_i generates a random number r_2 (a nonce) and calculates a series of hash values, $h(r_1, r_2, k_i^p[0])$, $h(r_1, r_2, k_i^p[1])$, ..., $h(r_1, r_2, k_i^p[d\text{-}1])$, $h(r_1, r_2, k_i^l)$, where $h(r_1, r_2, k)$ denotes the output of the hash function on three inputs: a key k and two random numbers r_1 and r_2. T_i replies R with a message $U = (r_2, h(r_1, r_2, k_i^p[0]), h(r_1, r_2, k_i^p[1]), ..., h(r_1, r_2, k_i^p[d\text{-}1]), h(r_1, r_2, k_i^l))$. For simplicity, we denote the elements in U as u, v_0, v_1, …, $v_{d\text{-}1}$,v_d where $u = r_2$ and $v_j = h(r_1, r_2, k_i^j)$, $j = 0...d\text{-}1$, $v_d = h(r_1, r_2, k_i^l)$. In the third phase, R identifies T_i using the key tree S and the received U.

Algorithm 2: Identification (U, node X)

```
1:  SUCCEED ← false;
2:  m ← DepthOfNode(X);
3:  IS ← GetIndexSet(X);
4:  IN ← |IS|;
5:  if m ≠ d
6:        for i = 1 to IN
7:              if v_m = h(r_1, r_2, i) ∧ i ∈ IS
8:                    Y ← GetChild(X,i);
9:                    Identification (U, Y);
10: else if m = d ∧ h(r_1,r_2, k_l) = v_d
11:       SUCCEED ← true;
12: if (SUCCEED = false)
13:       Fail and output 0;
14: Accept and output 1;
```

Reader R invokes a recursive algorithm to probe a path from the root to a leaf in S to identify T_i as shown in Algorithm 2. Assume R reaches a non-leaf node a at level m-1. For all index numbers stored in a, R computes the hash values with inputs r_1, r_2, and the index numbers. R then compares the hash values with the element v_m in the received U. If there is a match, the path of T_i should be extended to the child related to the index number. Note that here the child node is on the path assigned to T_i. Repeating such a procedure until arriving at a leaf node, R recognizes the tag T_i. For the example shown in Fig. 2, upon receiving a "Request" message with a random r_1, T_1 generates a random number r_2, and computes a series of hash values $h(r_1, r_2, 2)$, $h(r_1, r_2, 5)$, $h(r_1, r_2, 7)$, and $h(r_1, r_2, k_1^l)$, then replies R with the message $U = (u, v_0, v_1, v_2) = (r_2, h(r_1, r_2, 2), h(r_1, r_2, 5), h(r_1, r_2, 7), h(r_1, r_2, k_1^l))$. After receiving U, R computes all $h(r_1, r_2, x)$ to compare with v_1. Here $x = 2$, 5, and 7, which are all the index numbers stored in the root. Clearly, R locates 2 as a match number and thereby moves to node a. Then R locates 5 and 7 in the nodes b and node c, respectively. R terminates its path probing when it reaches the leaf node c, and hence identifying T_1.

E. Key-Updating

After successfully identifying T_i, R and T_i automatically update the key stored in T_i and coordinate the changes to the tree S as follows.

R makes use of a cryptographic hash function h to generate new keys. Let k_i^p and k_i^l be the current path key and leaf key used by T_i. R computes a new path key $\overline{k_i^p}$ from the old path key k_i^p and leaf key k_i^l by computing $\overline{k_i^p} = h(r_1, r_2, k_i^p, k_i^l)$. Similarly, R calculates the new leaf key as $\overline{k_i^l} = h(r_1, r_2, k_i^l)$.

The challenging issue here is that we need to carefully modify the index numbers of non-leaf nodes according to the new key $\overline{k_i^p}$. Otherwise, some tag identifications can be interrupted, since the index number stored in non-leaf nodes might be shared among multiple tags.

To address the challenge, we design two algorithms for key-updating: TagJoin shown in Algorithm 1 and TagLeave shown in Algorithm 3. The basic idea is that we first use the TagLeave to remove the path corresponding to old path key k_i^p of tag T_i, and then generate a new path corresponding to key $\overline{k_i^p}$ in S. It is possible that a non-leaf node in the path has multiple branches so that some keys are used by other tags, for example, node a in Fig. 2. In this case, the TagLeave algorithm terminates.

Algorithm 3: TagLeave (Tag T, Key Tree S)

```
1:  k^p, k^l ← GetKey(T);
2:  (k^p[0],…, k^p[d-1]) ← KeyDivision(k^p);
3:  Node ← GetLeaf(T);\\ Get the corresponding leaf node of T
4:  for i = d – 1 to 0
5:      if Node doesn't have brothers
6:          TempNode ← Node;
7:          Node ← FindParent(TempNode);
8:          Delete the k_i from the Index Set IS of Node;
9:          Delete TempNode;
10:     else Node←FindParent(Node);
```

After deleting the old key, R re-generates a new path for tag T_i according to the new key $\overline{k_i^p}$ using the TagJoin algorithm. A potential problem of new path generation is that the path has existed in S, which means the key $\overline{k_i^p}$ has been generated in the system. The probability of this situation happening is quite small. First, the sparse tree is a virtual tree according to the initialization algorithm. Prior to the tag deployment, the tree is empty. When a path key is generated by a hash function, a path from a certain leaf to the root emerges accordingly in the sparse tree. Therefore, a path in the sparse tree corresponds to a hash value. This correspondence leads to two facts: 1) the capability of a sparse tree is as large as the size of the hash value space. In our work, a path key is a hash value with a length of 128 bits, which indicates the sparse tree can hold 2^{128} paths at its maximum, that is, the sparse tree can hold 2^{128} tags correspondingly. In any practical RFID system, however, the number of tags is much less than 2^{128}. The probability that the tree becomes dense is negligible. 2) A path in the sparse tree corresponds to a hash value. Therefore, if two tags have the same path in the sparse tree, a hash collision appears. According to the collision-resistance property of hash functions, the probability of a hash collision happening is also negligible. For example, an RFID system contains 2^{20} tags, and the length of a path key is 128 bits. The ratio of occupied paths in the sparse tree is 2^{-88} ($2^{20}/2^{128}$), and the path key is generated uniformly at random. Thus, the probability of generating an existing path is 2^{-88}. It is safe to claim that the probability of two tags having a similar path is negligible based on the above analysis.

If such a collision does happen, in this design, R first generates a new key $\overline{\overline{k_i^p}}^2 = h(r_1, r_2, \overline{k_i^p}, k_i^l)$, and then executes the TagJoin algorithm again to create a new path in S. R repeats such a procedure until a new path is successfully generated. R counts the number that TagJoin runs, denoted as s (due to the negligible probability of collisions, s usually equals to 1), and sends a synchronization message $\sigma = (s, h(r_1, r_2, \overline{k_i^p}^s), h(r_1, r_2, \overline{k_i^l}))$ to tag T_i, as shown in Fig. 3. Here $\overline{k_i^p}^s$ is computed from iterative equations by:

$$\begin{cases} \overline{k_i^p}^1 = k_i^p \\ \overline{k_i^p}^s = h(r_1, r_2, \overline{k_i^p}^{s-1}, k_i^l) \end{cases} \quad (1)$$

Having σ, T_i can coordinate its key with the one generated by the reader. T_i first computes $\overline{k_i^p}^s$ using k_i^p and s with equation (1), then computes $\overline{k_i^l} = h(r_1, r_2, k_i^l)$. Thus, T_i gets $\sigma' = (s, h(r_1, r_2, \overline{k_i^p}^s), h(r_1, r_2, \overline{k_i^l}))$. After computing σ and σ', T_i verifies whether σ is identical to σ'. If yes, T_i updates its keys as $\overline{k_i^p}^s$ and $\overline{k_i^l}$ to finish the synchronization. Otherwise, T_i returns an error to the user.

F. System Maintenance

This component is mainly for tag joining and leaving. If a new tag T_i joins the system, R needs to find a new path in the key tree by invoking the TagJoin algorithm. In detail, R generates a new path key k_i^p and leaf key k_i^l independent to other keys, then splits k_i^p into d parts, $k_i^p[0], k_i^p[1],\ldots, k_i^p[d\text{-}1]$. Starting from the root, R constructs the path downwards. If R arrives at a non-leaf node j at level m, R adds $k_i^p[m]$ into j's index number set IS_j, and walks to the $k_i^p[m]$-th child of j (if this child does not exist, R creates it). When a leaf node is reached, R associates T_i to the leaf node, and sets the key of the leaf node as k_i^l. A new path is generated for T_i.

To withdraw a tag T_i, R should erase the path from the root to T_i's associated leaf node by using the TagLeave algorithm. Starting from the associated leaf node of T_i, R removes the path upwards. At the beginning, R deletes the leaf key k_i^l of T_i. If R reaches a node e at level m, R first finds e's parent f, and then deletes $k_i^p[m]$ from the index set IS_f. After arriving at node f, R deletes e. R repeats this procedure until a non-leaf node in the path has multiple branches, for example, node a in Fig. 2. Thus, R withdraws T_i.

IV. EFFICIENCY

We first investigate the storage efficiency of ACTION, and then analyze the identification efficiency by estimating the necessary number of hash computations. We also discuss the lower and upper bounds of ACTION's identification efficiency.

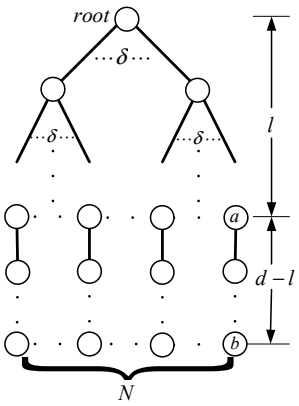

Figure 4. The worst case of ACTION.

A. Storage

An RFID tag normally has a very tiny memory for storing user information as well as the keys. Hence, storage efficiency must be taken into account in designing secure PPA protocols.

Without loss of generality, we assume that keys used by PPA protocols have an identical length, for example 64 bits, for security consideration. In balance-tree based approaches, each tag is allocated multiple keys, which incur a relatively large storage overhead. ACTION is more efficient in the key storage on both the tag and reader sides. On the tag side, ACTION allocates each tag only two keys, a path key and a leaf key. On the reader side, each path key is divided into several fractions and stored in the non-leaf nodes' index sets. Thus, R only stores $2N$ keys. In contrast, balance-tree based approaches distribute $O(\log_\delta N)$ keys to each tag, and maintain $\delta \times N$ keys on the reader side, where δ is the branching factor of the balance key tree. Therefore, ACTION is more practical for current RFID systems.

B. Identification Efficiency

The basic operations in a PPA authentication are hash computations and comparisons. The numbers of these two operations are equal, because each hash computation is followed by a comparison of hash values. Hence, we use the number of hash computations to estimate the time complexity. We present the best and worst cases in ACTION's authentication procedure, which are the computational lower bound and upper bound, respectively.

In the best case, the reader always meets a non-leaf node with only one index number at each level in the key tree. After d steps probing, the reader successfully identifies a tag. With the same branching factor setting δ, the depth of sparse tree is larger than the balance-tree, that is, $d > \log_\delta N$. Therefore, the computational lower bound of ACTION's identification is $\log_\delta N$.

As we assume the branching factor of the key tree is δ, each non-leaf node has at most δ children. In the worst case, at the root, the reader will compute δ hash values, and narrow the search scope to N/δ tags; at a child node of the root, the reader performs δ hash computations again. Then the reader narrows the search scope to N/δ^2 tags. At each level, the reader conducts δ hash computations. The reader repeats the same process at each level. At a given level l, the reader narrows the search scope to N/δ^l tags, and performs $l \cdot \delta$ hash computations. We assume at level l, the reader finds $N/\delta^l = 1$, or $l = \log_\delta N$. Since $d > \log_\delta N$, the reader does not reach leaf nodes at level l. We assume that the reader reaches a non-leaf node a at level l. The node a must have only one child (if a has two children, the number of tags in the system must be N+1, not N). Similar to a, each node of a's offspring has only one child. Thus, the reader will perform $d - l$ hash computations below the level l. We illustrate the worst case in Fig. 4.

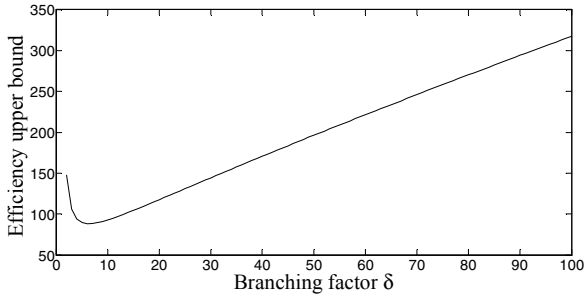

Figure 5. Efficiency upper bound vs. branching factor. (Assume $N = 2^{20}$, n = 128)

We calculate hash computations in the worst case, $f(\delta) = \delta \times l + d - l$. Since $l = \log_\delta N$, and $d = n/\log\delta$ (see Section III. *C*). We get

$$f(\delta) = \delta \cdot \log_\delta N + (\frac{n}{\log \delta} - \log_\delta N) = (\delta - 1) \cdot \log_\delta N + \frac{n}{\log \delta} \quad (2)$$

In equation (2), n is the bit length of keys in the system; in ACTION, n = 128. Let E_{ACTION} denote the efficiency of identification. We get

$$\log_\delta N < E_{ACTION} \le (\delta - 1) \cdot \log_\delta N + \frac{n}{\log \delta}$$

Thus, E_{ACTION} is $O(\log_\delta N)$. We plot the curve of the efficiency upper bound $f(\delta)$ in Fig. 5.

To find the optimal δ, we set $f'(\delta) = 0$. We get

$$\log_\delta N = \frac{n \log e \ln \delta}{(\delta(\ln \delta - 1) + 1) \log^2 \delta} \quad (3)$$

By solving equation (3), we find that δ = 8 is the optimal setup for identification efficiency. The upper bound is $f(8) = \frac{7}{3}\log N + \frac{n}{3}$. According to the relation between δ and d, if δ = 8, then d = 128/(log 8) = 128/3. By that setup, d is not an integer. Hence, in ACTION, we set a sub-optimal δ = 16, such that d = 32. The upper bound is $f(16) = \frac{15}{4}\log N + \frac{n}{4}$. Combined with the early discussion, we can see that the time complexity of ACTION authentication is $O(\log N)$.

V. Security Analysis

The essential goal of ACTION is to protect the privacy and defend against both passive and active attacks. These attacks are always the most important concerns in wireless environments [24-26]. For RFID systems, passive attack often

means eavesdropping on the communication between tag T and reader R, which are intensively discussed in previous designs [5-10, 13-19]. Active attackers can forge, replay, or discard the messages exchanged between T and R, so the attacks include tracking, cloning, and tag-compromising [7]. Attackers are even able to execute bogus authentication procedures. As ACTION has most of the advantages of early designs, it is inherently able to defend against passive attacks. Hence, in this section, we focus more on the ACTION's resistance to the compromising attack, which is the most serious threat to RFID systems but not addressed by early studies.

A. Compromising Attack

We present an attack model to formalize the capability of adversaries based on Avoine's [22] model. We create an interactive game G for two participants: an adversary A (the attacker) and a challenger C (the RFID system). Any attack to the RFID system can be represented as A's querying on one of C's oracles as follows:

Query(T, m_1, m_3): A sends a request m_1 to T. Subsequently, A receives a response from T. R then sends the message m_3 to T. Note that m_1 and m_3 represent the messages sent from A in the first and third rounds in an ACTION authentication procedure, respectively.

Send(R, m_2): A sends a message m_2 to R and receives a response. Note that m_2 represents the message sent from A in the second round in an ACTION authentication procedure.

Execution(T, R): A acts as a man-in-the-middle and executes an instance of the authentication protocol P with T and R, respectively. A then modifies the response messages and relays them to both sides accordingly.

Reveal(T): A compromises T, which means A obtains T's keys. Note that A can distinguish any given tag T from other tags if it can obtain T's keys.

Based on these oracles, the detailed procedure of game G between A and C consists of the following steps.

1. A interacts with C by accessing the above four oracles in polynomial time. In fact, adversary A performs a learning procedure on the system.

2. A informs C that the game has begun. C chooses two normal tags T_0 and T_1.

3. Let O_{T_0} and O_{T_1} denote the sets of accessed oracles of T_0 and T_1, respectively. For T_0 and T_1, A accesses their oracles in O_{T_0} and O_{T_1}.

4. C first accesses O_{T_0} and O_{T_1}, and then selects a bit $b \in \{0,1\}$ uniformly at random. C then provides the oracles of the corresponding tag T_b (if b=0, T_b =T_0, otherwise T_b=T_1) to A for access. For simplicity, we denote T_b as T. A then accesses T's oracle (except the *Reveal* oracle). Let the set of accessed oracles of T be O_T.

5. Based on the results from O_{T_0}, O_{T_1}, and O_T, A outputs a bit b'. If b'=b, A successfully distinguishes T_0 and T_1; otherwise, A loses. Note that A can access the oracles in O_{T_0}, O_{T_1} and O_T in polynomial time. Since T_0 and T_1 are chosen uniformly at random from all normal tags, if A can distinguish T_0 from T_1 (or vice versa) successfully with a probability non-negligibly larger than 1/2, this means that A can track all tags in an RFID system. Otherwise, we call that such an RFID system is private under the compromising attack.

Definition 1. A protocol P is private under the compromising attack, if for any polynomial time adversary A, the probability of A guessing b successfully under the above attack model satisfies:

$$\Pr[b' = b] \le \tfrac{1}{2} + 1/\,poly(s)\,.$$

Where $poly(s)$ are arbitrary polynomials, and s is a security parameter.

Based on Definition 1, for a given PPA based protocol P, we define the *advantage* of a compromising adversary by

$$\mathrm{Adv}_P(A) = \Pr[b' = b] - \tfrac{1}{2}$$

The advantage is a measurement of how successful an adversary can distinguish a tag from others. In our model, we assume that only two tags are normal and all other tags have been compromised. In this case, if the adversary can distinguish a normal tag from another one with a probability larger than 1/2, by using the obtained keys from compromised tags, we claim that the adversary has the advantage, and the value is determined by the difference between the probability and 1/2.

B. Defending Against Compromising Attacks

Based on the model, we formally prove that ACTION protocol is private under the compromising attack, which means an adversary has a negligible advantage when it conducts compromising attacks on the ACTION.

Theorem 1. Let q_Q, q_S, and q_E be the number of queries to the *Query*, *Send* and *Execute* oracles, respectively. ACTION is private under the compromising attack, and the advantage of a compromising adversary A is bounded by

$$\mathrm{Adv}_{\mathrm{ACTION}}(A) \le \frac{((d+1)(q_Q + q_E) + 4(q_E + q_S))^2}{2^{n+1}},$$

even if all other tags, except T_0 and T_1, in the system have been compromised by A, where d is the number of the key parts (or the depth of the key tree), and n is the bits number of a hash value as aforementioned in Section III. *C*.

Proof: In this proof, we use the *random oracle* (RO) model [23], in which hash functions are treated as arbitrary random functions. Since all keys in ACTION are generated independently, the keys of a tag are not related to those in other tags. We denote the game between the challenger C and the adversary A as G_0.

We introduce another challenger C', (who plays a simulated game G_1 with the adversary A), to simulate the real challenger C, and make them indistinguishable to A. Thus, from the viewpoint of A, the game G_1 between A and C' simulates exactly the real game G_0 between A and the real challenger C. On the other hand, we construct C' without the knowledge of T_0 and T_1's secret keys, k_0 and k_1. Thus, there is no information about k_0 and k_1 leaked to adversary A, so that A

must randomly guess which tag T_0 or T_1 is, that is, guessing the bit b (see the step 5 of the attack model in Section V. A) at random. In this case, the probability of a correct guess is 1/2. According to the definition of the compromising attack (refer to Section V. A), A's advantage in G_1 is 0. G_1 almost perfectly simulates the real game G_0, so the activities of the challenger C' would also perfectly simulate the real challenger C. However, without the knowledge of T_0 and T_1's secret key, there are some differences, called *Exceptions*, between the activities of C' and C in some situations. If we can estimate the probability of the *Exception* happening, we can compute the upper bound of A's advantage.

In G_1, the challenger C' simulates the hash function h in ACTION as a RO h'. The h' is constructed as a hash value list, *H_list*, maintained by C'. *H_list* is initialized as empty. The format of each item in *H_list* is (r_1, r_2, k, v), where v is the hash value of r_1, r_2, and k, i.e. $v = h(r_1, r_2, k)$.

For a query (r_1, r_2, k): If it exists in *H_list*, C' returns the corresponding $v = h(r_1, r_2, k)$; Otherwise C' picks up a v uniformly at random, returns the v as the answer of $h(r_1, r_2, k)$, and adds (r_1, r_2, k, v) into the *H_list*.

In the real game G_0, each message is computed with the hash function; the outputs of oracles *Query, Send,* and *Execute* are also computed with the hash function. Thus, we use the h' given above to construct the *Query, Send*, and *Execute* oracles in the G_1.

According to the ACTION protocol, the inputs of the *Query* oracle are "Request" and a nonce r_1, and the outputs are the authentication messages $U = (r_2, h(r_1, r_2, k_i^p[0]), h(r_1, r_2, k_i^p[1]),\ldots, h(r_1, r_2, k_i^p[d\text{-}1]))$. In G_1, the challenger C' simulates the *Query* oracle as follows:

Upon receiving the "Request" and r_1, the challenger C' first generates a nonce r_2 and two n-bits long keys k^p and k^l uniformly at random respectively, and then divides k^p into d parts, $k^p[0]$, $k^p[1]$,…, $k^p[d\text{-}1]$. Next, C' accesses the random oracle h' for d times to get the hash value sequence $h(r_1, r_2, k^p[0])$, $h(r_1, r_2, k^p[1])$,…, $h(r_1, r_2, k^p[d\text{-}1])$. Later on, C' computes the hash value $h(r_1, r_2, k^l)$ by accesses h'. Finaly, C' returns $U = (r_2, h(r_1, r_2, k^p[0]), h(r_1, r_2, k^p[1]),\ldots, h(r_1, r_2, k^p[d\text{-}1]), h(r_1, r_2, k^l))$.

Similarly, C' simulates the *Send* oracle in G_1 as follows:

Upon $U = (r_2, h(r_1, r_2, k_i^p[0]), h(r_1, r_2, k_i^p[1]),\ldots, h(r_1, r_2, k_i^p[d\text{-}1]), h(r_1, r_2, k_i^l))$: Generates a nonce r_1, a path key k^p, and a leaf key k^l uniformly at random; Accesses the random oracle h' to get the hash value $h(r_1, r_2, k^p, k^l)$ and $h(r_1, r_2, k^l)$; Accesses the random oracle h' to get the hash value $h(r_1, r_2, h(r_1, r_2, k^p))$ and $h(r_1, r_2, h(r_1, r_2, k^l))$. Returns $\sigma = (1, h(r_1, r_2, h(r_1, r_2, k)), h(r_1, r_2, h(r_1, r_2, k^l)))$ (where 1 is the value of s, the number of TagJoin algorithm running; see Section III. E).

Based on the *Query* and *Send* oracles, C' constructs the *Execute* oracle. Note that according to the definition of the *Execute* oracle, adversary A acts as a man-in-middle attacker between reader R and tag T. Therefore, in the procedure of performing *Execute* oracle, there are two stages: A communicates with T, which can be abstracted as *Query* oracle, and A communicates with R, which can be abstracted as *Send* oracle. Hence, the *Execute* oracle can be considered as the combination of *Query* and *Send* oracles. An access to *Execute* can also be regarded as an access to *Query* oracle plus an access to *Send* oracle. We thereby treat q_E *Execute* queries as q_E *Query* and q_E *Send* oracle accesses, respectively.

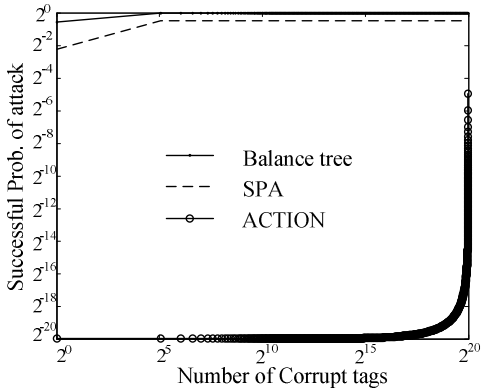

Figure 6. Comparisons on defending against the compromising attack (Assume $N = 2^{20}$).

G_1 is same as G_0, except the constructions of the *Query*, *Send* and *Execute* oracles. In G_1, from the viewpoint of A, C' simulates C perfectly except one event happens: a hash collision occurs in the output of the hash function. When the RO h' receives (r_1, r_2, k) and (r_2, r_1, k) as inputs, the two outputs should be identical in G_0. In G_1, however, the outputs of h' would not be identical. Thus, C' cannot answer A's query correctly. We define such a situation as an *Exception*. The probability of an *Exception* happening is bounded by the birthday paradox:

$$\Pr[\ Exception\ ] \le \frac{((d+1)(q_E+q_Q)+4(q_S+q_E))^2}{2\cdot 2^n} \qquad (4)$$

Note that the $(d+1)(q_E+q_Q)$ denotes the number of accesses to h'. Those accesses to h' are generated by q_E+q_Q *Query* oracle accesses, and each *Query* oracle access includes $d+1$ accesses to h'. Since in each *Send* oracle access, there are four accesses to h', the number of accesses to h' in accessing the *Send* oracle is $4(q_E+q_S)$, where n is the number of bits of a hash value, and d is the number of the divided parts of each tag's key.

As discussed at the beginning of the proof, the advantage of A in G_1 is zero. Therefore, considering the probability of *Event* happens, the advantage of the compromising adversary is bounded by:

$$\mathrm{Adv}_{\ \mathrm{ACTION}}\ (A) \le \frac{((d+1)(q_Q+q_E)+4(q_E+q_S))^2}{2^{n+1}}$$

Based on the Definition 1, ACTION is private under the compromising attack. Proved.■

Theorem 1 states that, under the attack model defined in Section V. A, the advantage of compromising attackers is negligible. Since q_E, q_Q, and q_S are polynomial to n, the advantage of compromising attackers approximates 0. That is, in the extreme case, even if a compromising attacker has captured $N-2$ tags, the probability of distinguishing a normal tag from another one is still 1/2. In general, assume the attacker has tampered with t tags. To distinguish a normal tag,

the attacker has to perform random guessing on $N - t$ normal tags, and the probability of correctly guessing, so the probability of a successful attack is $1/(N - t)$.

We compare the successful probabilities of compromising attacks in balance-tree based approaches, SPA, and ACTION. In this comparison, we assume SPA and balance-tree based approaches use binary trees. The RFID system contains 2^{20} tags. As shown in Fig. 6, in SPA and other balance-tree based approaches, compromising attackers have an overwhelming probability of distinguishing any normal tag after they tamper with 10 tags in the system, while ACTION perfectly eliminates the impact of compromising attacks.

We note that the path key of a tag may suffer from the key extracting attack. In ACTION, we set the branching factor as 16 when the length of a path key is 128-bits long. The path key will be divided into thirty-two 4-bit parts with this setting. In this case, for any identification message $h(r_1, r_2, k_i^p[j])$, an adversary can easily extract $k_i^p[j]$ by enumerating all 4-bit strings, like a brute-force search. Repeating the enumeration, the adversary can retrieve the entire path key. To fix this flaw, we introduce a leaf key k_i^l in the path key updating procedure. The length of each leaf key is similar to that of the path key, that is, 128 bits in our protocol. After identification, the path key is updated by $h(r_1, r_2, k_i^p, k_i^l)$ and the key k_i^l is also updated accordingly in each key updating procedure. Without knowing the leaf key, the adversary cannot predict the updated path key by guessing or performing a brute-force-like search on its sub-parts. Thus, ACTION can be resilient to an extracting attack.

VI. CONCLUSIONS

We propose a PPA protocol, ACTION, to support secure and efficient authentication in RFID applications. To the best of our knowledge, this is the first work that is able to defend against a compromising attack using tree-based approaches. The advantages of this design also include high efficiency in terms of storage and identification. We believe wide deployment of this design will make PPAs more practical and effective for large-scale RFID systems.

ACKNOWLEDGEMENTS

This work was performed at Hong Kong University of Science and Technology while Li Lu and Jinsong Han were Post-Doc Fellows and Renyi Xiao was a Visiting Scholar. Li Lu is now an assistant professor with School of Computer Science & Engineering, University of Electronic Science & Technology of China, and Renyi Xiao is with National Natural Science Foundation of China. This work is supported in part by the Hong Kong GRF grant HKUST6169/07E and Hong Kong ITF GHP/044/07LP. Yunhao Liu and Jinsong Han are supported in part by the NSFC/RGC Joint Research Scheme N_HKUST 602/08, NSFC grant No. 60873262, National High Technology Research and Development Program of China (863 Program) under grant No. 2007AA01Z180, and NSFC Key Project grants No. 60533110 and No. 60736016.

REFERENCES

[1] T. Kriplean, E. Welbourne, N. Khoussainova, V. Rastogi, M. Balazinska, G. Borriello, T. Kohno, and D. Suciu, "Physical Access Control for Captured RFID Data," *IEEE Pervasive Computing*, vol. 6, 2007.

[2] B. Sheng, C. C. Tan, Q. Li, and W. Mao, "Finding Popular Categories for RFID Tags". in Proceedings of ACM Mobihoc, 2008.

[3] Y. Li and X. Ding, "Protecting RFID Communications in Supply Chains," in Proceedings of ASIACCS, 2007.

[4] P. Robinson and M. Beigl, "Trust Context Spaces: an Infrastructure for Pervasive Security in Context-Aware Environments," in Proceedings of International Conference on Security in Pervasive Computing, 2003.

[5] S. Weis, S. Sarma, R. Rivest, and D. Engels, "Security and Privacy Aspects of Low-Cost Radio Frequency Identification Systems," in Proceedings of International Conference on Security in Pervasive Computing, 2003.

[6] T. Dimitriou, "A Secure and Efficient RFID Protocol that Could make Big Brother (partially) Obsolete," in Proceedings of PerCom, 2006.

[7] D. Molnar and D. Wagner, "Privacy and Security in Library RFID: Issues, Practices, and Architectures," in Proceedings of CCS, 2004.

[8] D. Molnar, A. Soppera, and D. Wagner, "A Scalable, Delegatable Pseudonym Protocol Enabling Owner-ship Transfer of RFID Tags," in Proceedings of Selected Areas in Cryptography - SAC, 2005.

[9] L. Lu, J. Han, L. Hu, Y. Liu, and L. M. Ni, "Dynamic Key-Updating: Privacy-Preserving Authentication for RFID Systems," in Proceedings of PerCom, 2007.

[10] G. Avoine, E. Dysli, and P. Oechslin, "Reducing Time Complexity in RFID Systems," in Proceedings of Selected Areas in Cryptography - SAC, 2005.

[11] S. Bono, M. Green, A. Stubblefield, A. Juels, A. Rubin, and M. Szydlo, "Security Analysis of a Cryptographically-Enabled RFID Device," in Proceedings of USENIX Security, 2005.

[12] M. C. O'Connor, "Taking Advantage of Memory-Rich Tags", http://www.rfidjournal.com/magazine/article/2925.

[13] A. Juels and S. Weis, "Defining Strong Privacy for RFID," in Proceedings of PerCom, Workshop PerTec, 2007.

[14] A. Juels, "RFID Security and Privacy: a Research Survey," *Journal of Selected Areas in Communications*, vol. 24, 2006.

[15] M. Ohkubo, K. Suzuki, and S. Kinoshita, "Efficient Hash-Chain based RFID Privacy Protection Scheme," in Proceedings of UbiComp, Workshop Privacy, 2004.

[16] A. Juels, "Minimalist Cryptography for Low-Cost RFID Tags," in Proceedings of International Conference on Security in Communication Networks - SCN, 2004.

[17] T. Dimitriou, "A Lightweight RFID Protocol to Protect Against Traceability and Cloning Attacks," in Proceedings of SecureComm, 2005.

[18] G. Tsudik, "YA-TRAP: Yet Another Trivial RFID Authentication Protocol," in Proceedings of PerCom Workshops, 2006.

[19] G. Avoine and P. Oechslin, "A Scalable and Provably Secure Hash Based RFID Protocol," in Proceedings of PerCom, Workshop PerSec, 2005.

[20] M. E. Hellman, "A Cryptanalytic Time-Memory Trade-off," *IEEE Transactions on Information Theory*, vol. 26, 1980.

[21] "Expands Tag-it ISO/IEC 15693 RFID Product Line", News Releases from Texas Instruments, http://www.ti.com/rfid/shtml/news-releases-rel12-14-05.shtml.

[22] G. Avoine, "Adversarial Model for Radio Frequency Identification," Technical Report 2005/049. http://eprint.iacr.org/2005/049, 2005.

[23] M. Bellare and P. Rogaway, "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols," in Proceedings of CCS, 1993.

[24] M. Shao, Y. Yang, S. Zhu, and G. Cao, "Towards Statistically Strong Source Anonymity for Sensor Networks," in Proceedings of Infocom, 2008.

[25] W. Gu, Z. Yang, C. Que, D. Xuan and W. Jia, "On Security Vulnerabilities of Null Data Frames in IEEE 802.11 based WLANs, " in Proceedings of ICDCS, 2008.

[26] Z. Jiang, J. Wu, and D. Wang, "A New Fault-Information Model for Adaptive & Minimal Routing in 3-D Meshes," *IEEE Transactions on Reliability*, vol. 57, 2008.