DBSJ Journal, Vol.8, No.2
September 2009

An Efficient Identification and Indexing for Secure RFID with d-Left Hashing

Yasunobu NOHARA♥ Sozo INOUE♦

In this paper, we propose a secure identification scheme for RFID with efficient time and memory, and also an efficient update of pre-computed values on the server side. Although RFID (Radio Frequency IDentification) is becoming popular, a privacy problem still remains, where an adversary can trace users' behavior by linking identification log by legitimate/adversary readers. For this problem, a hash-chain scheme has been proposed as a secure identification for low-cost RFID tags, and its long identification time has been reduced by Avoine et al. using pre-computation on the server side. However, Avoine's scheme uses static pre-computation, and therefore pre-computed values include ones which are already used and no longer used. In this paper, we optimize a lookup of pre-computed values using d-left hashing, and provide efficient update of pre-computed values. We also show reasonable analytical result for memory and pre-computation / identification / update time.

1. Introduction

RFID (Radio Frequency IDentification) is a technology to identify humans and objects via radio frequencies. An RFID device, often called an *RFID tag*, includes a small IC chip with an ID and a radio frequency function. Every human and object to be identified is labeled with an RFID tag. An RFID tag transmits its ID to a reader via radio frequency. The tag's ID can be linked with related information, e.g. the location, time, etc., and stored in a database. RFID systems enable to relate people or objects in the real world with databases in the virtual world, and therefore are acknowledged as fundamental systems for pervasive computing.

However, RFID raises potential violation of personal privacy. If RFID tags are attached to personal belonging like shoes, watches, etc., then an adversary can collect users' personal information by reading these tags because these tags have strong relationships with each user. The leakage of this information leads two privacy problems.

1. Leakage of personal belonging information
2. Tracing a user's behavior by reading and linking identification log by legitimate and/or adversary readers

As for 2, *unlinkability* is a property that an adversary cannot recognize whether outputs are from the same user. Unlinkability is important for preventing identification logs to be traced by adversaries. Moreover, achieving unlinkability is highly challenging, since RFID tag's output has to be changed every time, which involves many system problems.

In this paper, we focus on unlinkability when the system performs identification. Identification is main functionality of RFID systems, and we denote identification which offers unlinkability as *secure identification*. Secure identification is completely different from authentication, since most authentication protocols firstly communicate the entity's plain and static ID to the other, while secure identification does not allow this to achieve unlinkability.

The hash-chain scheme [1,2] provides unlinkability against an adversary by using one-way hash functions. The scheme is suitable for RFID systems because the implementation cost of an RFID tag must be low in the system. However, the scheme is not scalable since the server needs $O(N)$ hash calculations for every identification where N is the number of RFID tags.

For this problem, Avoine et al. improved the hash-chain scheme using time-memory trade-off [3]. In their method, they compress the result of pre-computation, and utilize it for identification. They achieved reduction of memory size and identification time. However, their method has a problem in the update of pre-computation. Since their method performs pre-computation statically, the method gradually includes values already used or never to be used when identifications are repeated. To remove these wasted values, reconstructing the pre-computation and keeping minimal pre-computed values only for the near future identification is needed. However, achieving this with Avoine's method is very costly, since it has to do pre-computation with the same cost as the initial pre-computation. Thus, to design a secure identification scheme with efficient update as well as efficient pre-computation is a crucial challenge in large-scale RFID systems.

In this paper, we propose the solution using d-left hashing, which is a time/memory-efficient lookup technique. We also show analytic comparison with existing methods, and show that our method is the most efficient for the memory and time for identification / pre-computation / update.

2. Hash-chain Scheme and Challenges

2.1 Overview of RFID Systems

An RFID system consists of RFID tags and the server(s). An RFID tag is attached to people, objects. The *server* consists of multiple *readers*, which communicate with

♥ Non-member Faculty of Information Science and Electrical Engineering, Kyushu University
nohara@irvs.is.kyushu-u.ac.jp
♦ Member Faculty of Engineering, Kyushu Institute of Technology sozo@mns.kyutech.ac.jp

RFID tags, and computation abilities. Readers are often distributed in wide area.

In the rest of the paper, for simplicity, we use the term "server" to represent any readers as if there is only single reader. However, we have to be aware that the data on the server which is given from the communication with an RFID tag might be obsolete.

2.2 Hash-Chain Scheme

The hash-chain scheme, proposed by Ohkubo et al. [1,2], is one of the schemes that provide unlinkability against an adversary by using one-way hash functions. In this scheme, two different one-way hash functions H and G, a ROM, and a non-volatile memory are embedded to each RFID tag. Let N be the number of RFID tags in an RFID system. And L denotes the length of each output of one-way hash functions H and G.

At the time of shipping, each RFID tag i $(1 \le i \le N)$ is stored a bit string id_i into the ROM, and secret information $s_{i,1} \in \{0,1\}^L$ into the non-volatile memory. We assume that if $i \ne i'$, then $id_i \ne id_{i'}$ for $1 \le i, i' \le N$. Moreover, the server stores the pair $(id_i, s_{i,1})$ of all tags.

For unlinkability, the secret information changes on every identification, and an output is filtered, each of which using the different hash functions. Precisely, the j-th$(1 \le j \le M_{max})$ output $o_{i,j}$ and the secret information, denoted as $s_{i,j}$, of RFID tag i are given as:

$$o_{i,j} = H(id_i || s_{i,j})$$
$$s_{i,j+1} = G(s_{i,j})$$

where $||$ means the concatenation of strings.

In an identification process, we need to execute a hash inverse calculation since id_i is calculated from the hashed value $o_{i,j}$. Here, we define two naive methods for secure identification aiming to be compared with our proposed method in the later sections.

One is to calculate $H(id_i||s_{i,j})$ for every identification process. The other is to pre-compute all the results $H(id_1||s_{1,1}), \cdots, H(id_1||s_{1,M_{max}}), H(id_2||s_{2,1}), \cdots, H(id_N||s_{N,M_{max}})$, and store them to a memory (or transparently, storage) on the server. We call the former method a *sequential search*, and the latter a *LUT (Look Up Table) search*. The successful rates of identification by both the sequential search and the LUT search are 100%.

The output $o_{i,j} = H(id_i||s_{i,j})$ of an RFID tag i is not fixed because $s_{i,j}$ changes on every identification. It is difficult to get id_i from $o_{i,j}$ due to the property of one-way hash function H. Therefore, this scheme provides unlinkability against an adversary, while the server has the ability of identification.

Moreover, from the definition of one-way hash function G, it is also difficult to obtain $s_{i,j'}$ $(j' < j)$ even if id_i and $s_{i,j}$ are tampered with. Therefore, the scheme provides *forward security*, meaning that no RFID tag can be traced to be linked from past ID information even if the secret information in the device is known to an adversary.

2.3 Problem of Hash-chain Scheme

The existing search method of the hash-chain scheme has to perform NM_{max} one-way hash calculations on average (sequential search) or to prepare $NM_{max}(L + \log N)$ [bit] memory (LUT search). Therefore, it is difficult to apply an existing hash-chain scheme to a large-scale system, where NM_{max} is large.

For this problem, Avoine et al. propose speeding up of the hash-chain scheme on the server side using time-memory trade-off [3]. The proposed method is to compress the pre-computed values, and perform identifications using the compressed values. The method uses the rainbow table, which is a lookup table offering a time-memory trade-off used in recovering the plaintext password from a password hash [4]. It is effective for minimizing memory usage and identification time, while the successful rate of identification does not reach to 100% due to the nature of time-memory trade-off. Their method performs pre-computation statically, and it gradually includes values already used or never to be used as described below, when identifications are repeated.

Since identification process is executed on a first-come-first-served basis in general, $o_{i,j}$ $(1 \le j \le l_i)$ should be excluded from the pre-computed values, where l_i is the index of the tag i's last output that the server receives [2]. Moreover, the probability that the server receives $o_{i,j}$ $(l_i + M + 1 \le j \le M_{max})$ is negligible if an adequate margin M, which is discussed in Section 4.5, is chosen. Therefore, the minimal pre-computed values for the next single identification are $o_{i,j}$ $(l_i + 1 \le j \le l_i + M)$, and other values are wasted. The identification problem against the minimal pre-computed values is easier than the original identification problem; however, reconstructing the pre-computation is needed for every identification process since the minimal pre-computed values change dynamically.

However, Avoine's scheme has to completely reconstruct the rainbow table with a change of the pre-computation values due to the nature of the rainbow table. Therefore, Avoine's original (static) scheme needs extra memory without heavy update calculation, and Avoine's dynamic scheme needs heavy update calculation with minimal memory. Therefore, to design a secure identification scheme with efficient pre-computation and its efficient update is a crucial challenge in large-scale RFID systems.

3. d-left Pre-computation Scheme

In this section, we propose a d-left pre-computation scheme, which realizes secure identification with efficient updates. The proposing scheme utilizes d-left hash table [5,6], which is an efficient hash table in memory size and memory accesses, as a basic block. At first, we explain the d-left hash table and describe an idea of d-left hash table for hash inverse calculation. Then, we propose the d-left pre-computation scheme.

3.1 Background: d-left Hash Table

A d-left hash table is one of data structures using d independent hash functions [5]. The hash table has m buckets, and each bucket can store n records at maximum. Let $B[i]$ be the i-th bucket in the hash table and $|B[i]|$ be the number of the records stored in $B[i]$. Each hash function h_i $(1 \le i \le d)$ maps some set element to one of the m bucket positions with a uniform random distribution. Fig.1 shows an example of a d-left hash table with parameters $d = 2, m = 8, n = 5$.

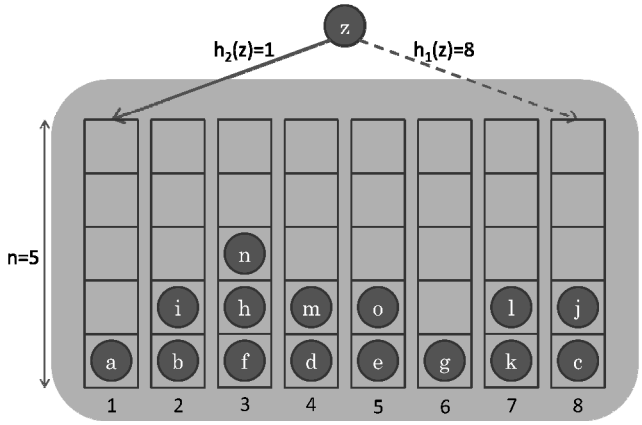

Fig.1 Example of d-Left Hash Table (d=2, m=8, n=5)

To add a record, feed it to each of the d hash functions to get the d bucket positions. The incoming record is placed in the bucket containing the smallest number of records; in case of a tie, the record is the placed into the left-most bucket. The asymmetry introduced by breaking ties toward the left actually improves performance, in that the maximum number of items placed in a bucket is smaller (in a probabilistic sense) when one breaks ties in this manner [5]. In Fig.1, hash functions generate two bucket position $h_1(z) = 8$ and $h_2(z) = 1$. The number of the records of the 1st bucket $|B[1]| = 1$ is smaller than that of the 8th bucket $|B[8]| = 2$. Therefore, z is inserted into the 1st bucket $B[1]$.

In order to retrieve a record in the hash table, feed it to each of the d hash functions to get d bucket positions. Then, the contents of d buckets are checked. Since the d-left hashing balances the bucket load, the table can achieve lookup speed with better worst-case performance in practice [7]. To delete a record from the hash table, search the record to determine which bucket stores the record. Then, delete the record from the bucket.

To avoid using pointers to reduce memory size, the bucket size n is fixed. The fixed bucket size may result overflow; however, the probability of overflow is quite low since the d-left hashing balances the bucket load [6].

3.2 d-left Hashing for Hash Inverse Calculation

In this subsection, we describe a novel idea to reduce a memory usage of the d-left hash table. A usually lookup table, including a d-left hash table must store a pair of a key z and a value x as a record. However, there exists a relation $z = f(x)$ for hash inverse calculation $x = f^{-1}(z)$. Thus, the key z can be calculated from the value x.

In our method, a set of indexes X that can be identified by the server is given as:

$$X = \{(i,j) | 1 \le i \le N,\ l_i + 1 \le j \le l_i + M\}$$

The stored data are modified from the pair of (z,x) to x, where $x \in X$ and $z \in \{0,1\}^L$. Since $N \ll 2^L$ as mentioned, it is $\log|X| \ll L$, and stopping storing makes a large reduction of memory usage. Since our scheme does not store z in the table, we need to calculate z from x for every identification process. However, we only calculate at most dn records in the d buckets. We can explain the above idea as follows.

We firstly reduce the candidates to a small constant number using d-left hashing, and then check the candidates in a constant time. More formally describing, given $z \in \{0,1\}^L$, we calculate $X' \subseteq X$ which includes $x = f^{-1}(z)$ for given $z \in \{0,1\}^L$ using d-left hashing, where $|X'|$ is almost a constant. Then, we check at most dn elements of X' if each of them matches $f^{-1}(z)$. If the match exists, the server returns the value as x. Otherwise, the function returns $\emptyset$ since there are no matches.

3.3 d-left Pre-computation Scheme

The proposed d-left pre-computation scheme has three phases, 1) pre-computing phase, 2) identification phase, and 3) update phase. The pre-computing phase is done only once. The identification phase and the update phase are executed for each identification process.

Our scheme is similar to a Bloom pre-computation scheme [8]. The Bloom pre-computation scheme uses a Bloom filter [9]. Since the Bloom filter cannot erase a record, an update phase of our scheme is quite different from that of the Bloom pre-computation scheme.

In the proposed scheme, the hash table B and the last index l_i of a tag i are stored in a memory on the server.

3.3.1 Pre-computing Phase

In pre-computing phase, the server stores an initial output set $\{o_{i,1}, \cdots, o_{i,M}\}$ for each tag in the d-left hash table. When a new record is stored in the d-left hash table, the record is inserted into the least loaded bucket. The server stores only a pair of (i,j) and z is not stored in the hash table. Since record insertions of the d-left hash table may fail, we discuss this problem in the Section 4.1.

Algorithm 1 shows the procedure of pre-computing phase. $InsertHC(i,j)$ is an algorithm which stores $o_{i,j}$ in the d-left hash table, and described in Algorithm 2. $MakeOutput(i,j)$ is a function to obtain $o_{i,j}$.

Algorithm 1 Pre-computing Phase

```
1: for i = 1 to N do
2:     l_i ← 0
3:     for j = 1 to M do
4:         InsertHC(i, j)
5:     end for
6: end for
```

Algorithm 2 $InsertHC(i,j)$

Input: $i \in \{1, \cdots, N\},\ j \in \{1, \cdots, M_{max}\}$

```
1: z ← MakeOutput(i, j)
2: k ← arg min_{l=1,...,d} {|B[h_l(z)]|}
3: if |B[h_k(z)]| < n then
4:     Insert (i, j) to B[h_k(z)]
5: else
6:     return `Insertion fails'
7: end if
```

3.3.2 Identification Phase

This phase, shown in Algorithm 3, searches x from given z, such that $x \in X, z = f(x)$. In case of $x \in X, z = f(x)$, x must be stored in one of the buckets $B[h_l(z)]$, where $1 \le l \le d$. Therefore, the server returns x' such that $f(x') = z, x' \in X'$ if exist, where $X' = \cup_{l=1}^{d} B[h_l(z)]$. Otherwise, the server returns $\emptyset$.

Algorithm 3 Identification Phase

Input: $z \in \{0,1\}^L$

Output: $x \in X \cup \{\emptyset\}$

```
1: X' ← ⋃_{l=1}^{d} B[h_l(z)]
2: for all (i,j) ∈ X' do
3:     if MakeOutput(i,j) = z then
4:         return (i,j)
5:     end if
6: end for
7: return ∅
```

3.3.3 Update Phase

In update phase, shown in Algorithm 4, the server updates the tag i's last index l_i and the hash table using (i,j) which can be obtained in the above identification phase.

Algorithm 4 Update Phase

Input: $i \in \{1,\cdots,N\}$, $j \in \{1,\cdots,M_{max}\}$

```
1: for k = l_i + 1 to j do
2:     z ← MakeOutput(i,k)
3:     for l = 1 to d do
4:         if (i,k) ∈ B[h_l(z)] then
5:             Delete (i,k) from B[h_l(z)]
6:         end if
7:     end for
8:     InsertHC(i,k+M)
9: end for
10: l_i ← j
```

An output set of the tag i that is stored in the server before and after update phase is given as follows:

$$Y_i^{old} = \left\{o_{i,l_i^{old}+1}, \cdots, o_{i,l_i^{old}+M}\right\}$$
$$Y_i^{new} = \left\{o_{i,j+1}, \cdots, o_{i,j+M}\right\}$$

Therefore, the relationship of the two sets is given as

$$Y_i^{new} = Y_i^{old} \cup \left\{o_{i,l_i+M+1}, \cdots, o_{i,j+M}\right\} - \left\{o_{i,l_i^{old}+1}, \cdots, o_{i,j}\right\}.$$

At first, the server deletes $\left\{o_{i,l_i^{old}+1}, \cdots, o_{i,j}\right\}$ from the hash table. To delete a record (i,j) that is representing the j-th output of the tag i, the server calculates $o_{i,j}$ using $MakeOutput(i,j)$. The record (i,j) must be stored in the one of the buckets $B[h_l(o_{i,j})]$, where $1 \le l \le d$. Therefore, the server finds the bucket that stores the record (i,j), and delete the record from the bucket.

Secondly, $\left\{o_{i,l_i+M+1}, \cdots, o_{i,j+M}\right\}$ is inserted into the hash table. $InsertHC(i,j)$, described in the pre-computing phase, is used to insert the record (i,j) that representing the $o_{i,j}$. Finally, the server updates $l_i \leftarrow j$.

4. Analysis

In this section, we analyze our proposed scheme quantitatively.

4.1 Memory Usage

In pre-computation phase, NM entries for tag output $o_{i,j}$ are stored. Although these data are modified on every identification process, d-left hash table always stores NM entries since the same number of additions and removals are done.

The d-left hash table has m buckets, and each bucket has at most n entries. Therefore, it needs mn space. In the space, pairs of (i,j) are stored. Since the possible pairs of (i,j) are $1 \le i \le N, 1 \le j \le M_{max}$, it needs $\log NM_{max}$ [bit] memory for one (i,j) pair. Therefore, the hash table needs $dmn \cdot \log NM_{max}$ [bit] memory size. It does not require pointers, since the size of buckets is fixed.

The memory usage and the possibility of saturating the bucket depend on the parameters d, m and n. Bonomi et al. showed the parameters $d = 4, m = NM/6, n = 8$ gives the possibility of saturating the bucket is 1.681×10^{-27}, even if we perform 2^{20} times of addition/removal over NM entries, which is small enough to negligible[6]. In the rest of the paper, we adopt the above value as d, m and n. Thus, the memory usage of our scheme becomes $\frac{4}{3} NM \log NM_{max}$[bit].

4.2 Time of Pre-computation

In a pre-computation phase, NM entries for RFID tag outputs $o_{i,j}$ for any (i,j) are stored. A single output of an RFID tag requires 2 hash calculations. Therefore, $2NM$ hash calculations are done.

4.3 Identification Time

For single identification, exhaustive hash calculations are done for the buckets $X_{h_l(x)}$, where $1 \le l \le d$. The expected number of the records stored in these d buckets is $dNM/m = 24$, and we need $(24+1)/2$ calculations of $MakeOutput(i,j)$ on average. Since the calculation is not a sequential such as $o_{i,j}, o_{i,j+1}, \cdots$, the average cost for one $MakeOutput(i,j)$ calculation is $M/2 + 1$. Therefore, the time for identification is $12.5(M/2 + 1)$ on average.

4.4 Update Time

In the phase of updating, the data entries are added as the same number as deletion. For single addition/deletion of entry, it requires two hash calculations for each single output calculation. If we assume j of $o_{i,j}$ is distributed randomly from $l_i + 1$ to $l_i + M$, $M/2$ entries are replaced on average. Therefore, the average hash calculation is $2M$.

Table.1 Comparison of Required Memory and Time

	Memory[bit]	Average Number of Hash Calculation			Successful Rate of Identification[%]
		Identify	Update	Pre-comp	
LUT	$NM_{max}(L+\log N)$	0	0	$2NM$	100
Avoine(Static)	$cNM_{max}L$	$\frac{108(\log NM_{max})^2}{c^3L^2}$	0	$\frac{NM_{max}^2}{2}$	99.9
Avoine(Dynamic)	$cNML$	$\frac{108(\log NM)^2}{c^3L^2}$	$\frac{NM^2}{2}$	$\frac{NM^2}{2}$	99.9
d-left	$\frac{4NM}{3}\log NM_{max}$	$12.5(\frac{M}{2}+1)$	$2M$	$2NM$	100
Bloom	$\frac{NM}{\ln 2}\log \varepsilon^{-1}$	$(\varepsilon N+1)M$	$2M$	$2NM$	100
Sequential	0	NM	0	0	100

4.5 Size of Margin

We discuss the relationship between a margin for synchronization M and a synchronization problem by limiting the search range.

A *synchronization problem* is an error that the server cannot follow the transition of the secret information of the tag, and cannot identify the tag. In the hash-chain scheme, the tag outputs only $o_{i,j}$ and does not output the secret information $s_{i,j}$, described in Section 2.2. Since we need $s_{i,j}$ for finding id_i from given $o_{i,j}$, it is important to estimate $s_{i,j}$ efficiently.

If the server can observe all outputs of all tags, we can determine s_{i,l_i+1} is the secret information of tag i, where o_{i,l_i}, is the tag i's last output that the server receives. However, several reasons, e.g. 1) there are many readers, including adversaries' that the server cannot know, 2) there are many communication errors in an RFID system, prevent the server from reading all outputs of all tags. Then, we estimate the secret information as $s_{i,l_i+1},\cdots,s_{i,l_i+M}$. If the number of the reading failures is less than M, the server can keep synchronization. Since fewer M often occurs synchronization problem, we should choose adequate M.

M should be increased in case 1) an adversary can access the tag easily, 2) a communication error often occurs, and 3) an application is damageable against synchronization problem. On the other hand, high frequency service by the server is a reason to decrease the M, and smaller M is desirable from the viewpoint of the server cost. Since we should consider various kinds of conditions, quantitative determination of M is a future challenge.

In an environment in which M should be large, that is, there are many reading failures of the server, M_{max} also should be large to keep the maximum number of identification. Therefore, we assume $M_{max} \gg M$.

4.6 Comparison with Related Work

Table 1 shows the results of the analytical evaluation. Analysis results of Avoine's scheme [3] and Bloom pre-computation scheme [8] is cited from each papers. We set a parameter γ of Avoine's scheme as 8 and Successful Rate of Identification (SRI) of the Avoine's scheme becomes 99.9%

Since our scheme stores minimal pre-computation values for next single identification, our scheme uses fewer memory than Avoine's static method, where $M_{max} \gg M$. And our scheme can reduce the computation cost of update than Avoine's dynamic method, where $NM \gg 1$ since our scheme uses the modified d-left hashing, which can update data easily. Moreover, our scheme can save a memory than both of Avoine's schemes since hash table only stores values and keys are not stored in the hash table, where $2^L \gg N$. Since modified d-left hashing is more space-efficient than a Bloom filter, our scheme needs fewer memory than Bloom pre-computation scheme.

Both of Avoine's schemes and the Bloom pre-computation scheme can change memory usage by changing the parameters c or ε, and these schemes can adopt for various memory constraints; however a memory usage of our scheme is automatically determined by N and M, thus our scheme has a week point from this point.

5. Conclusion

In this paper, we proposed a secure identification scheme for RFID with efficient time and memory, and also an efficient update of pre-computed values on the server side.

Our d-left pre-computation scheme is based on the following ideas.

- Storing the minimal pre-computation values in a memory for next single identification to improve memory efficiency
- Introducing a modified d-left hash table for efficient update of pre-computed values and using the table for reducing the candidates of identification

While our scheme achieves fast identification, the scheme also achieves same unlinkability and forward security as Ohkubo's original works. We also showed reasonable analytical evaluation for memory and pre-computation / identification / update time.

[Acknowledgement]

This work has been supported by Grant-in-Aid for Scientific Research (A) (21680009) and Grant-in-Aid for Scientific Research on Priority Areas (21013038). We are

grateful for this support.

[Reference]

[1] M. Ohkubo, K. Suzuki and S. Kinoshita: "Cryptographic approach to a privacy friendly tag", RFID Privacy Workshop@MIT (2003).
[2] M. Ohkubo, K. Suzuki and S. Kinoshita: "Hash-chain based forward-secure privacy protection scheme for low-cost RFID", 2004 Symposium on Cryptography and Information Security - SCIS2004, Vol. 1, pp. 719-724 (2004).
[3] G. Avoine and P. Oechslin: "A scalable and provably secure hash-based RFID protocol", 2nd International Workshop on Pervasive Computing and Communications Security - PerSec2005, IEEE Computer Society Press, pp. 110-114 (2005).
[4] P. Oechslin: "Making a faster cryptanalytic time-memory trade-off", Advances in Cryptology - CRYPTO 2003, Vol. 2729 of LNCS, pp. 617-630 (2003).
[5] A. Broder and M. Mitzenmacher: "Using multiple hash functions to improve IP lookups", Proceedings of 20th Annual Joint Conference of the IEEE Computer and Communications Societies - INFOCOM 2001, Vol. 3, pp. 1454-1463 vol.3 (2001).
[6] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh and G. Varghese: "An improved construction for counting Bloom filters", 14th Annual European Symposium on Algorithms - ESA 2006, Vol. 4168 of LNCS, pp. 684-695 (2006).
[7] H. Song, S. Dharmapurikar, J. Turner and J. Lockwood: "Fast hash table lookup using extended Bloom filter: an aid to network processing", SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications, ACM Press, pp. 181-192 (2005).
[8] Y. Nohara, S. Inoue and H. Yasuura: "A secure high-speed identification scheme for RFID using Bloom filters", Proc. of 3rd International Conference on Availability, Security and Reliability - ARES2008, IEEE Computer Society, pp. 717-722 (2008).
[9] B. H. Bloom: "Space/time trade-offs in hash coding with allowable errors", Communications of the ACM, 13, 7, pp. 422-426 (1970).

Yasunobu NOHARA

Yasunobu Nohara is a post-doctoral fellow at the Faculty of Information Science and Electrical Engineering, Kyushu University. He received his B.E., M.E., and Ph.D. degrees in Computer Science from Kyushu University. His current research focuses on the privacy and security of RFID systems and human/object tracking for a robot's activity. He is a member of IPSJ, IEICE, and IEEE.

Sozo INOUE

Sozo Inoue is an associate professor in the Faculty of the Basic Science, Kyushu Institute of Technology. His research interests include ubiquitous computing systems and applications, particularly security and privacy in RFID systems, and medical applications. Inoue was born in 1974 and received his Doctor of Engineering from Kyushu University. He is a member of IPSJ, the Database Society of Japan (DBSJ), ACM, and the IEEE Computer Society.