A Secure High-Speed Identification Scheme for RFID Using Bloom Filters

Nohara, Yasunobu
Graduate School of Information Science and Electrical Engineering, Kyushu University

Inoue, Sozo
Kyushu University Library

Yasuura, Hiroto
Faculty of Information Science and Electrical Engineering, Kyushu University

出版情報：Proceeding of the 3rd International Conference on Availability, Security and Reliability, pp.717-722, 2008-03-05
バージョン：
権利関係：

A Secure High-Speed Identification Scheme for RFID Using Bloom Filters

Yasunobu NOHARA, Sozo INOUE, Hiroto YASUURA
Kyushu University
744 Motooka Nishi-ku Fukuoka 819-0395 JAPAN
{nohara,sozo,yasuura}@c.csce.kyushu-u.ac.jp

Abstract

As pervasive computing environments become popular, RFID tags are introduced into our daily life. However, there exists a privacy problem that an adversary can trace users' behavior by linking the tag's ID. Although a hash-chain scheme can solve this privacy problem, the scheme needs a long identification time or a large amount of memory. In this paper, we propose an efficient identification scheme using Bloom filters, which are space-efficient probabilistic data structures. Our Bloom pre-calculation scheme provides a high-speed identification with a small amount of memory by storing pre-calculated outputs of the tags in Bloom filters.

1 Introduction

RFID (Radio Frequency IDentification) is a technology to identify humans and objects through *RFID tag*; that is, silicon chips with IDs and radio frequency functions. As pervasive computing environments become more commonplace, RFID tags are becoming part of our daily life.

Privacy is a serious concern when RFID is used. For example, the *location privacy problem* – the possibility that an adversary can trace a user's behavior by reading and linking a tag's ID – is a privacy issue.

Unlinkability is a property which means an adversary cannot recognize whether outputs are from the same user, and this property is important with respect to the privacy problem. The *hash-chain scheme* [1, 2] provides unlinkability against an adversary by using one-way hash functions. However, the scheme needs a long identification time or a large amount of memory.

In this paper, we propose an efficient identification scheme using Bloom filters. A *Bloom filter* is a space-efficient probabilistic data structure that is used to test whether an element d is a member of a set X [3]. The filter can separate effectively any element that is not in the set, with small error probability. Our *Bloom pre-computation scheme* provides a high-speed identification with a small amount of memory by storing pre-calculated outputs of the tags in Bloom filters.

The remainder of this paper is organized as follows. Section 2 describes the hash-chains scheme and defines the identification problem. Section 3 explains the Bloom filter. Section 4 proposes the Bloom pre-computation scheme. Section 5 compares our scheme with other scheme. Section 6 concludes this paper.

2 Hash-chain Scheme

2.1 Overview

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 within each RFID tag. Let N be the number of RFID tags in an RFID system where the ID id_i of an RFID tag i is a bit string of length l for $1 \leq i \leq N$. We assume that if $i \neq j$, then $id_i \neq id_j$ for $1 \leq i, j \leq N$, and $2^l \gg N$. For $s, t \in \{0,1\}^*$, we denote by $s\|t$ the concatenation of s and t. Let L be the length of the one-way hash functions H and G.

The RFID tag i stores id_i in the ROM, and stores secret information $s_{i,1} \in \{0,1\}^L$ in the non-volatile memory. The server stores the pair $(id_i, s_{i,1})$ $(1 \leq i \leq N)$ of all tags.

The j-th output of the RFID tag i (denoted $o_{i,j}$) is given as follows:

$$o_{i,j} = H(id_i \| s_{i,j})$$

The RFID tag i updates the secret information as follows:

$$s_{i,j+1} = G(s_{i,j})$$

We assume $o_{i,j} = o_{i',j'} \iff i = i'$ and $j = j'$.

We suppose that for every tag i, the server maintains an expected counter value l_i s.t. $l_i + 1 \leq latest_i \leq l_i + M$, where $o_{i,latest_i}$ is the latest output of the tag i. An adequate M helps to prevent out of synchronism even if there

is several readers that cannot be controlled by the server. The server can find (i, j) corresponding to an output d by checking $d = H(id_i \| s_{i,j})$ for all $1 \leq i \leq N$ and all $l_i + 1 \leq j \leq l_i + M$. After the identification of $o_{i,j}$, l_i is updated to j.

There are two calculating policies in identification. One policy is calculating $H(id_i \| s_{i,j})$ for each identification. Another policy is using pre-calculated results $O_i = \{o_{i,l_i+1}, \cdots, o_{i,l_i+M}\}$, which are stored in a memory. We call the former identification policy *sequential search*. And we call the later one *LUT search*.

We define successful of identification as outputting correct i when $d \in O_i$ is given as challenge. And we define successful of denial as outputting $\perp$ when o is given, where $\forall i, o \notin O_i$. The successful rates of identification by the sequential search and the LUT search are 100%. And also their successful rates of denial are 100%.

$X = H(id_i \| s_{i,j})$ is not fixed because $s_{i,j}$ changes every time. It is computationally difficult to get id_i from X due to the property of one-way hash function. Therefore, this scheme provides unlinkability against an adversary.

2.2 Problems of Hash-chain Scheme

The existing search methods of the hash-chain scheme, has to perform NM one-way hash calculations in average(sequential search) or prepare NML[bit] memory(LUT search). Therefore, it is difficult to apply the existing hash-chain schemes to a large-scale system, where NM is large.

Now, we define the identification problem. Let f be a bijective function from a set X to a set Y. f is also an one-way hash function that is easy to compute but "hard to invert". Let A be a subset of X. Y is a subset of a set Z. We assume $|Y| \ll |Z|$.

We define a function $g^A : Z \to A$ is as follows.

$$g^A(z) = \begin{cases} a & \exists a \in A, z = f(a) \\ \perp & otherwise \end{cases} \quad (1)$$

The corresponding relations between the above formalization and the hash-chain scheme are as follows: $X \leftrightarrow \{(i, j)\}$, $Y \leftrightarrow \{o_{i,j}\}$, $f \leftrightarrow H(id_i \| s_{i,j})$ and $Z \leftrightarrow \{0, 1\}^L$. Then, we can translate the identification problem of the hash-chain scheme into the problem how to construct an algorithm and data structure that have two properties given as follows:

1. $g^A(z)$ can be computed efficiently with lower memory
2. The data structure can be changed efficiently when A is changing to $A' \subset X$

Changing from A to A' is corresponding to a change O_i owing to an update of l_i.

Table 1. List of Notation

Symbol	Short Description
H, G	One-way hash function
N	Number of RFID tags in the system
M	Margin for synchronization
$o_{i,j}$	j-th output of the RFID tag i
l_i	Expected counter value of the tag i
O_i	Output set of the tag i
L	Output length of one-way hash functions
m	Length of an array of a Bloom filter
k	Number of hash functions using a filter
n	Number of the elements stored in a filter
ϵ	Probability of false positives of a filter
$BLOOM_i$	Bit array of the Bloom filter that stores O_i
d	Input
c	Memory parameter

2.3 Notation

Table 1 shows the symbols that are used in this paper. At the first opportunity, we explain the other symbols that are not explained in this section.

3 Bloom Filter

The Bloom filter, proposed by Burton H. Bloom, is a space-efficient probabilistic data structure that is used to test whether an element d is a member of a set X [3]. The Bloom filter outputs 'Positive' if the filter concludes d is a member of X; otherwise the filter outputs 'Negative'.

The Bloom filter never yields *false negative error*, which the filter outputs 'Negative' in spite of $d \in X$. However, the filter may yields *false positive error*, which the filter outputs 'Positive' in spite of $d \notin X$.

3.1 Algorithm

The data structure of a Bloom filter is a bit array of m bits. $BLOOM[i]$ denotes the i-th element of the array $BLOOM$. The Bloom filter uses k independent random hash functions, $h_1, h_2, \cdots, h_k$ with range $[1, m]$. h_i doesn't need the 'one-way' property.

3.1.1 Initialization

To clear the set X, all the bits of $BLOOM$ are set to 0. Algorithm 1 shows the initialization process written in pseudo-code.

Algorithm 1 $InitBloom(BLOOM)$

Input: $BLOOM$
Output: $BLOOM$
1: **for** $i = 1$ to m **do**
2: $\quad BLOOM[i] \leftarrow 0$
3: **end for**

3.1.2 Insertion

To insert an element d into the set X, the $BLOOM[h_i(d)]$ are set to 1 for $1 \leq i \leq k$. Algorithm 2 shows the insertion process written in pseudo-code.

Algorithm 2 $AddBloom(d, BLOOM)$

Input: $d, BLOOM$
Output: $BLOOM$
1: **for** $i = 1$ to k **do**
2: $\quad BLOOM[h_i(d)] \leftarrow 1$
3: **end for**

3.1.3 Query

To check whether an element d is a member of the set X, we check whether all $BLOOM[h_i(d)]$ are set to 1 for $1 \leq i \leq k$. If not, it is clear that d is not a member of X. If all $BLOOM[h_i(d)]$ are set to 1, we assume that d is a member of the set X although the filter may yield a false positive error.

Algorithm 3 shows the query process written in pseudo-code.

Algorithm 3 $QueryBloom(d, BLOOM)$

Input: $d, BLOOM$
Output: Positive or Negative
1: **for** $i = 1$ to k **do**
2: $\quad$ **if** $BLOOM[h_i(d)] = 0$ **then**
3: $\quad\quad$ **return** Negative
4: $\quad$ **end if**
5: **end for**
6: **return** Positive

3.2 Probability of false positive error

Let n be the number of elements in the set X. The probability of a false positive error (denoted ϵ) is given as follows.

$$\epsilon = [1 - (1 - 1/m)^{kn}]^k \approx (1 - e^{-kn/m})^k$$

For given m and n, we can find the value of k that minimizes the probability ϵ is

$$k = \frac{m}{n} \ln 2.$$

The minimized false positive error rate is given as follows.

$$\epsilon = 0.5^k \approx 0.6185^{m/n}$$

4 Bloom Pre-computation Scheme

In this section, we propose a *Bloom pre-computation scheme*, which realizes efficient identification using Bloom filters. The 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.

4.1 Pre-computing Phase

We prepare the N Bloom filters. Each Bloom filter is corresponding to each tag. $BLOOM_i$ denotes the filter corresponding to the tag i. The $BLOOM_i$ stores the output set O_i.

Algorithm 4 shows the procedures of pre-computing phase written in pseudo-code. $make\_output(i, j)$ is a function to obtain $o_{i,j}$.

Algorithm 4 Pre-computing Phase

Input: $BLOOM_1, \cdots, BLOOM_N, l_1, \cdots, l_N$
Output: $BLOOM_1, \cdots, BLOOM_N, l_1, \cdots, l_N$
1: **for** $i = 1$ to N **do**
2: $\quad l_i \leftarrow 0$
3: $\quad InitBloom(BLOOM_i)$
4: $\quad$ **for** $j = 1$ to M **do**
5: $\quad\quad id \leftarrow make\_output(i, j)$
6: $\quad\quad AddBloom(id, BLOOM_i)$
7: $\quad$ **end for**
8: **end for**

Since we need to calculate $o_{i,1}, o_{i,2}, \cdots, o_{i,M}$ for each tag, the number of the one-way hash calculations for pre-computing is $2NM$.

And since each Bloom filter stores O_i, the parameters n, m, k are given as follows: $n = M$, $m = n \log_{0.6185} \epsilon = M \log_{0.6185} \epsilon$ and $k = \log_{0.5} \epsilon$. Therefore, memory usage of the scheme is $mN = NM \log_{0.6185} \epsilon$[bit], and the number of random hash calculation h_i is $MNk = MN \log_{0.5} \epsilon$.

4.2 Identification Phase

To search id_i from d, we query all Bloom filters whether d is a member of the set. If d is the j-th output of the tag i, $BLOOM_i$ returns 'Positive'. Therefore we can obtain id_i by searching a filter that returns 'Positive'.

However, we may not determine the ID because some Bloom filters may return 'Positive' due to false positive

errors. To solve this problem, we execute the sequential search, described in section 2.1 for all the tags whose Bloom filters return 'Positive'. By sequential search, we can also obtain j.

Algorithm 5 shows the procedures of pre-computing phase written in pseudo-code.

Algorithm 5 Identification Phase

Input: $d, BLOOM_1, \cdots, BLOOM_N, l_1, \cdots, l_N$
Output: (i, j) or $\perp$

```
1: for i = 1 to N do
2:   if QueryBloom(d, BLOOM_i) = Positive then
3:     for j = l_i + 1 to l_i + M do
4:       if d = make_output(i, j) then
5:         return (i, j)
6:       end if
7:     end for
8:   end if
9: end for
10: return ⊥
```

The Bloom filters that output 'Positive', consist of one Bloom filter which stores $d \in X$ and any Bloom filters that yield false positive error. The expected number of the Bloom filter that outputs 'Positive' is $\epsilon(N-1)+1 \simeq \epsilon N+1$. Since we need to calculate the output $M/2$ times on average for each tag i, the expected number of the one-way hash calculation is $\{\epsilon(N-1)+1\}M \simeq (\epsilon N+1)M$.

The number of random hash calculation h_i is only $k = \log_{0.5} \epsilon$ because all Bloom filters use same h_i values in identification phase.

Since a Bloom filter never yields false negative error, the successful rate of identification is 100%. And since we execute sequential search for all the tags whose Bloom filters return 'Positive', the successful rate of denial is also 100%.

4.3 Update Phase

In update phase, we update l_i and $BLOOM_i$ using (i, j) which can be obtained in above identification phase.

Firstly, we update $l_i \leftarrow j$. Secondly, we clear the $BLOOM_i$ to delete all stored elements. Finally, we insert new O_i into the $BLOOM_i$.

Algorithm 6 shows the procedures of update phase of Bloom filter written in pseudo-code.

Since we need to generate $o_{i,j+1}, \cdots, o_{i,j+M}$ and store them into the $BLOOM_i$, the number of one-way hash calculations in update phase is $2M$ and the number of random hash calculation h_i is $Mk = M \log_{0.5} \epsilon$.

Algorithm 6 Update Phase

Input: $BLOOM_i, l_i, i, j$
Output: $BLOOM_i, l_i$

```
1: l_i ← j
2: InitBloom(BLOOM_i)
3: for k = l_i + 1 to l_i + M do
4:   id ← make_output(i, k)
5:   AddBloom(id, BLOOM_i)
6: end for
```

5 Comparison

In this section, we first introduce related work. After that, we compare the proposed schemes with existing schemes. Last, we show the implementation result of the proposed schemes.

5.1 Related Work

5.1.1 Bloomier Filter

The Bloomier filter is an extension to Bloom filter for compactly encoding a function to support approximate evaluation queries [4]. The problem definition of the Bloomier filter is similar to our problem definition. The difference between the Bloomier filter and the proposed scheme is as follows.

1. In a Bloomier filter, f^{-1} is given instead of f.
2. A Bloomier filter may not output $\perp$ in spite of $\forall a \in A, z \neq f(a)$.
3. Bloomier filter doesn't consider the update of A

5.1.2 Avoine's scheme

Avoine *et al.* [5] developed a specific time-memory tradeoff that reduces the amount of computation in the hash-chain scheme. The purpose of the Avoine's scheme is same as that of the proposed scheme.

However, the Avoine's scheme needs to calculate one-way hash function $O(NM^2)$ times in pre-computing process. And there is a problem that pre-computing process is needed when A is updated because the scheme doesn't consider the update of A.

Reducing the frequency of update decreases the load of update; however the virtual margin for synchronization M also reduces. Therefore, the tolerance against out of synchronism becomes weaker. In addition, there is a security risk in that an adversary can guess a tag's count number [6].

In Avoine's scheme, the successful rate of identification is less than 100% although the successful rate of denial is 100%.

5.2 Comparison

Table 2 compares the memory and the time needed for each scheme. In the table, c is the memory size parameter for Avoine's scheme, and SRI is stand for Successful Rate of Identification. Note that the memory amount does not include the space for the ID list, which is required for every scheme. The number of random hash calculations h_i on Bloom filters is given in brackets. The calculation load of the random hash functions h_i is lighter that that of the one-way hash functions H and G because h_i doesn't need one-way property.

5.3 Implementation

We implemented the sequential search scheme, the LUT search scheme, the Avoine's scheme and the proposed scheme. We set N is 102400, and M is 256. We vary the memory usage of the proposed scheme.

The following is the execution environments of the server.

- OS: CentOS5 (kernel-2.6.18)
- CPU: AMD Athlon64 3000+
- Memory: 1GB
- Compiler: GCC 4.1 with O2 option

We adopted SHA-1 [7], whose output length L is 160bit, as the one-way hash functions H and G.

Table 3 shows the implementation results for execution time.

Compared with the proposed scheme and the LUT scheme, our scheme (except for 25MB) can identify with faster time and lower memory although our scheme need longer time for pre-computing. Also the total execution time (=identification+update) of our scheme is faster than the LUT scheme.

The total time of our scheme is about 1000 times faster than that of the sequential scheme.

Compared with our scheme and the Avoine's scheme, the identification time of the Avoine's scheme is faster than that of our scheme under the same memory usage. However, the total execution time of our scheme is about 10000 times faster than that of the Avoine's because the Avoine's scheme needs heavy calculation for updating. Therefore, our scheme is more suitable for environments in which adversary can access the tag easily because our scheme can always keep the constant margin for synchronization.

Compared with our schemes varying memory usage, 87.5MB, not 100MB is the best memory usage which gives the fastest total executing time. We can recognize this result as follows. The large memory usage reduces the expected number of the one-way hash calculation H and G. On the other side, the large memory usage increases the number of random hash calculation h_i. Therefore, the usage of 87.5MB memory gives the best balance between the load of the one-way hash functions H, G and the load of the random hash calculations h_i.

6 Conclusion

In this paper, we proposed a Bloom pre-computation scheme, which realizes efficient identification using Bloom filters. Our scheme provides a high-speed identification with a small amount of memory by storing pre-calculated outputs of the tags in Bloom filters.

Compared with the Avoine's scheme, our scheme can update the pre-calculation results efficiently and can always keep the constant margin for synchronization. Our scheme is more suitable for environments in which adversary can access the tag easily.

References

[1] M. Ohkubo, K. Suzuki, and S. Kinoshita, "Cryptographic approach to a privacy friendly tag," in *RFID Privacy Workshop@MIT*, Nov. 2003.

[2] M. Ohkubo, K. Suzuki, and S. Kinoshita, "Hash-chain based forward-secure privacy protection scheme for low-cost RFID," in *2004 Symposium on Cryptography and Information Security – SCIS2004*, vol. 1, pp. 719–724, Jan. 2004.

[3] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, pp. 422–426, Jul. 1970.

[4] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The bloomier filter: an efficient data structure for static support lookup tables," in *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, (Philadelphia, PA, USA), pp. 30–39, Society for Industrial and Applied Mathematics, 2004.

[5] G. Avoine and P. Oechslin, "A scalable and provably secure hash-based RFID protocol," in *2nd International Workshop on Pervasive Computing and Communications Security – PerSec2005*, pp. 110–114, IEEE Computer Society Press, Mar. 2005.

[6] A. Juels and S. A. Weis, "Defining strong privacy for RFID," in *IACR Cryptology ePrint Archive Report*, no. 2006-137, 2006.

[7] National Institute of Standards and Technology, Federal Information Processing Standards Publication 180-2, *SECURE HASH STANDARD*, 2002.

Table 2. Comperison of required memory and time

	Memory[bit]	Average Number of Hash Calculations			SRI[%]
		Identify	Update	Pre-comp.	
LUT	NML	0	M	$2NM$	100
Proposed	$NM \log_{0.6185} \epsilon$	$(\epsilon N + 1)M$ $[+\log_{0.5} \epsilon]$	$2M$ $[+M \log_{0.5} \epsilon]$	$2NM$ $[+NM \log_{0.5} \epsilon]$	100
Avoine	cNL	$\frac{108M^3(\log NM)^2}{c^3L^2}$	$\frac{21NM^2}{2c} + NM$	$\frac{21NM^2}{2c} + NM$	99.9
Sequential	0	NM	0	0	100

Table 3. Implementation Results for Execution time

	Memory[MB]	Identify[msec]	Update[msec]	Total[msec]	Pre-comp.[sec]	SRI[%]
LUT	500	119.1	0.108	119.2	22.9	100
Proposed	100	4.9	0.524	5.4	53.2	100
	87.5	4.7	0.482	5.2	48.9	
	75	4.8	0.455	5.2	46.0	
	62.5	5.4	0.410	5.8	41.7	
	50	10.1	0.368	10.5	37.4	
	37.5	37.8	0.325	38.1	33.1	
	25	246.4	0.297	246.7	30.2	
Avoine	180	0.1	1.23e5	1.23e5	122.7	100
	60	0.9	2.78e5	2.78e5	278.4	100
	30	5.2	5.52e5	5.52e5	552.0	100
	12	72.2	1.11e6	1.11e6	1111.2	96.1
	6	579.0	2.02e6	2.02e6	2021.8	94.2
Sequential	0	10614.7	0.001	10614.7	0.0	100