International Journal of Information Security (2022) 21:669–687
https://doi.org/10.1007/s10207-021-00575-2

REGULAR CONTRIBUTION

Gossamer: weaknesses and performance

P. D'Arco[1] · R. De Prisco[1] · Z. Ebadi Ansaroudi[1] · R. Zaccagnino[1]

Published online: 13 January 2022

Abstract
In this paper, we focus on Gossamer, a well-known ultralightweight authentication protocol, introduced in 2008. Our contributions are the following:

– we analyze the structure of the MixBits function, a key component of the protocol, and show that it does not realize a pseudorandom function, not even in a weak form;
– we show, by employing artificial intelligence techniques, that tags are distinguishable;
– finally, we study the performance of Gossamer and show that it does not provide a substantial saving, compared to a standard three-round mutual authentication protocol, implemented with lightweight primitives.

We close the paper with further comments and remarks.

Keywords Ultralightweight · Authentication protocols · Gossamer

1 Introduction

Ultralightweight authentication The process through which a party can convince another party of its identity, called *authentication*, is a fundamental process, in order to realize secure and private applications. Authentication protocols, by granting access to sensitive and valuable resources only to legitimate parties, enable organizations to protect their data and network infrastructures. They are built on something *the party is*, e.g, biometric authentication based on the iris or the fingerprint, or on something *the party knows*, e.g., authentication based on a password, a pin, a secret key, or on something *the party holds*, e.g., authentication based on a token, a credential, a physical device. In the past years, several techniques have been developed. However, what makes authentication still subject of investigations is the nature of the computational environment surrounding us nowadays. Heavily-constrained devices, like cheap sensors or tiny tags, cannot afford the computational efforts required by protocols designed for traditional communication networks and devices.

Computationally-light and storage-efficient solutions are needed.

To achieve the above goal, around 2006, some authentication protocols using very simple operations were presented. Such protocols are M^2AP [32], LMAP [31] and EMAP [30]. The hardware target they were addressing is represented by circuits with a few hundred gates, for example, the ones used in some applications of the RFID technology. However, all of them presented some weaknesses, which were used to set up efficient attacks. SASI [13], introduced the next year, designed for providing Strong Authentication and Strong Integrity, received considerable attention both from cryptanalysts and by designers. Chien, in [13], offered a sort of categorization of protocols, according to the tools they need. He used the term *full-fledged* to refer to the class of protocols requiring support for conventional cryptographic functions like symmetric encryption, hashing, or even public key cryptography. He used the term *simple* to refer to the class of protocols requiring random number generation and hash-

✉ R. De Prisco
robdep@unisa.it

P. D'Arco
pdarco@unisa.it

Z. Ebadi Ansaroudi
zebadiansaroudi@unisa.it

R. Zaccagnino
rzaccagnino@unisa.it

[1] Dipartimento di Informatica, University of Salerno, 84080 Fisciano, SA, Italy

ing, while he used the term *lightweight* to refer to the class of protocols which require random number generation and simple checksum functions. And, finally, he used the term *ultralightweight* to refer to the class of protocols which only involve simple bitwise operations, like *and, or, xor,* $+_{\bmod n}$, and *cyclic shift*.

SASI, as its predecessors, was quickly broken in a few months, e.g., [17,21,34,38], and more refined attacks followed, e.g., [6,18]. In some papers, warnings were raised against such ultralightweight solutions [5,8,18]. In [18], a full analysis of SASI was provided and, in general, the limits of such approaches, not based on sound security arguments, were stressed. In [5], a full guide to the common pitfalls, which are usually present in the design of ultralightweight authentication protocols, was provided to designers and practitioners. Unfortunately, adhoc protocols with informal security analysis continue to be presented at a considerable rate, and, not surprisingly, they are broken quickly after publication. We refer the interested reader to [16] for a look at some further examples of the weaknesses of some proposals of the last few years, and to [11,15] for an updated view of the current state of knowledge and of future perspectives for such a research area.

The only significant exception to the representation of the ultralightweight area we have just provided is the Gossamer protocol [33], whose security and privacy properties, more or less, have not been subverted. Its design is more involved compared to the designs of the other protocols. It uses a round function, MixBits, for mixing the input values, and it is closer in spirit to some designs of standard lightweight cryptographic primitives. Due to such a state of affair, we decided to give a further look at this protocol.

Research goals and methodology The security analysis is not trivial at all. Indeed, Gossamer has been already scrutinized in the past, by employing the cryptanalytic techniques applied in the area, without finding relevant weaknesses. Therefore, we decided to look first at its parts to evaluate, at the end, the whole: in the Gossamer case, the MixBits function is the most important part. Indeed, from a conceptual point of view, Gossamer can be seen as a protocol which uses, at each interaction, MixBits to produce new values from old ones, needed to compute the authentication messages and update the state information. Ideally, the MixBits function should behave like a pseudorandom function. Thus, our first research question was: *does MixBits approximate well a pseudorandom function?* From a theoretical point of view, a family of functions is pseudorandom if no efficient adversary, with oracle access to a device which implements either a truly random function or one of the functions from the family, chosen uniformly at random, is able to decide which one is the case. The adversary, in its attempt, can query the oracle with input values *of his choice*, receiving as replies, the output values of the function, evaluated on the provided input values. In practice, functions which should behave pseudorandomly, are structured in rounds of computation, and are designed to exhibit the *avalanche effect*. It basically means that, by looking at the executions of the function evaluation on two even slightly different input strings, say x and x', at each round of the computation, the differences between the round input strings increase and are spread over all the bits of the round output strings. In such a way, the final output strings, $y = f(x)$ and $y' = f(x')$, are radically different. The MixBits function follows such a design strategy. However, by using simple statistical tests, we show that MixBits does not perfectly approximate the avalanche effect. It follows, immediately, that an efficient distinguisher can be implemented to test whether a black-box implements MixBits or a truly random function. Therefore, our second research question was: *does MixBits approximate well a weak pseudorandom function?* Indeed, weak pseudorandomness only requires the non-existence of an efficient distinguisher, on inputs chosen *uniformly at random*. In such a case, we decided to employ machine learning techniques to see whether they are able to learn relations which hold among the outputs of the MixBits function. Our decision was motivated by two reasons: first, it seemed to us a nice and general way to look at the existence of relations. Then, we also thought that there are a few studies but not much literature on the use of machine learning techniques to achieve cryptanalytic objectives. Hence, our findings could have been of independent interest in the field, adding some elements of novelty, useful in other protocol analyses. The intuition was right since, with several techniques and different efficacy degrees, we show that MixBits does not approximate well a weak pseudorandom function, too. Then, the following natural research question was: *how much MixBits weaknesses impact on Gossamer?* Surprisingly, we notice that the other operations of the protocol compensate MixBits weaknesses. In particular, the C value which is used for authenticating the reader to the tag, at each iteration, is subject to a good avalanche effect. Hence, our simple statistical test for distinguishing it from a random value fails. This rules out basic attack strategies, usually applied against such protocols, in which an adversary tries to impersonate the reader by efficiently computing/guessing the authentication value in a protocol session. Therefore, in order to get a better understanding of the protocol design, we analyze a couple of simplified versions of the Gossamer protocol, obtained by removing some operations. In such a way, we notice and point out some interesting properties of the protocol. Then, our next research question was on the privacy side. Precisely, we asked: *does Gossamer guarantee basic privacy properties?* By using a simple privacy model and employing again machine learning techniques, we are able to show that tags can be distinguished with a non-negligible advantage over a truly random guessing. Finally, our last research question was: *how much efficient is Gossamer?* We choose a platform and compare Gossamer computational performance against the performances of a standard three-round mutual authentication protocol, implemented through lightweight primitives. With all the limits that our software comparison has, we point out that the computational effort required by Gossamer does not provide a substantial saving, compared to the lightweight implementations of the standard three-round mutual authentication protocol.

Organization of the paper In Sect. 2, we briefly introduce the system model we deal with, and describe the Gossamer protocol, explaining its rationale and its working mechanisms. Then, in Sect. 3 we provide some background on machine learning and we describe the techniques we have used, while, in Sect. 4, we start our analysis. We focus on the MixBitsfunction and show some features it exhibits under an adversarial action. More precisely, as anticipated before, by using two different techniques, we show that it does not implement either a pseudorandom function or a weak pseudorandom function. Then, in Sect. 5, we introduce a simple privacy model, based on the most common and used ones, and show that tags can be distinguished. Finally, in Sect. 6, we consider an implementation of Gossamer and provide a performance evaluation, in a comparative setting. We close the paper in Sect. 7, summarizing our findings and providing some further comments, hints and suggestions for future researches.

2 System and protocol specification

In this section, we describe the system with its components, and the Gossamer protocol, its implementation, and the messages exchanged between the protocol parties. We also discuss some known attacks.

2.1 System model

A Radio Frequency Identification (RFID, for short) system consists of three components: RFID tags, RFID readers and a back-end server. RFID readers and the back-end server are connected through a secure channel and are usually considered in the literature as a single entity. An RFID tag is a small resource-limited device, with restricted computational power and communication capability. It is uniquely identified by an authorized RFID reader via its identifier. However, without authentication, since the tag and the reader communicate over an insecure channel, e.g., radio waves, the tag's data can be easily tampered by an illegal third party. To protect the tag, an authentication protocol is needed.

An RFID mutual authentication protocol should satisfy three main properties:

– *Correctness* the legitimate tags are authenticated by the readers and vice versa
– *Security* an adversary cannot impersonate a legitimate tag to a reader and a legitimate reader to a tag
– *Privacy* tags do not leak sensitive information to an adversary.

We will specify a rigorous privacy model in Sect. 5. For more information about RFID technology we refer the reader to textbooks on the topic. Plenty of them are now available.

2.2 The protocol

The Gossamer protocol is an ultralightweight authentication protocol in which, a tag and a reader, exchange four messages to achieve mutual authentication. The protocol has two phases, one called *identification*, and the other one called *authentication*. In the first, the party provides information to be recognized in the system, in the second the party proves its claimed identity. The first two messages belong to the identification phase, while the last two belong to the authentication phase. An updating phase is needed upon a successful completion of the authentication phase, both on the tag and on the reader. Tags have a real name (static identifier), denoted with ID, which is never exposed. Instead, in the protocol, a pseudonym, denoted with IDS, is used. Moreover, two secret keys, k_1 and k_2, are shared by the tag and the reader. Both the tag and the reader need to store the triple (IDS, k_1, k_2). All these elements are strings of N bits, where N is fixed; for Gossamer is set to 96. At each interaction, a new IDS and new keys k_1, k_2 are used. When the interaction between tag and reader succeeds, the new triple of values (IDS', k_1', k_2') replaces the old one. The update $(IDS, k_1, k_2) \leftarrow (IDS', k_1', k_2')$ is performed by the tag, upon a successful processing of the third message, and by the reader, upon a successful processing of the fourth message. Since an interaction might partially succeed, resulting in an update only on the tag, the tag needs to store both the current/old triple of values and the potential new one, which will become the current one, upon a successful completion of an interaction. In such a way, the tag can roll back to the previous triple, if necessary.

The tag has to be able to perform the following operations: bitwise xor $(\oplus)$, modular addition $(+)$, and circular rotation (Rot). We denote with $\mathsf{Rot}(x, y)$ the left circular rotation of x by $(y \bmod N)$ positions.

The reader has to be able to generate random numbers: at each interaction, the reader generates two new N-bit random numbers, n_1 and n_2, sometimes called *nonces*, since they are used only once.

The protocol uses a bit scrambling function, called MixBits, that takes as input two N-bit numbers and produces

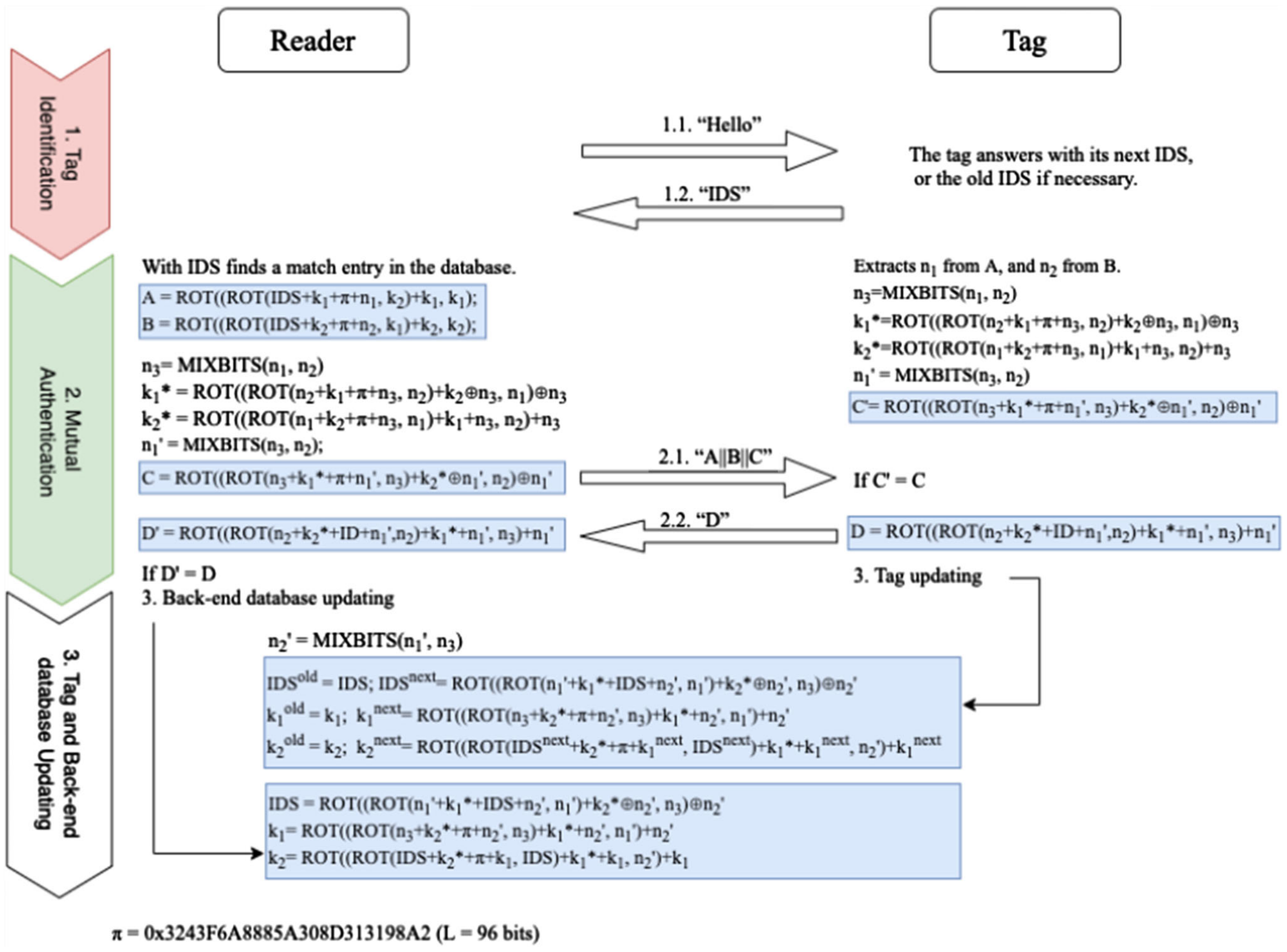
Reader
Tag
1. Tag Identification
2. Mutual Authentication
3. Tag and Back-end database Updating
1.1. "Hello"
1.2. "IDS"
The tag answers with its next IDS, or the old IDS if necessary.
With IDS finds a match entry in the database.
A = ROT((ROT(IDS+k1+π+n1, k2)+k1, k1);
B = ROT((ROT(IDS+k2+π+n2, k1)+k2, k2);
n3= MIXBITS(n1, n2)
k1* = ROT((ROT(n2+k1+π+n3, n2)+k2⊕n3, n1)⊕n3
k2* = ROT((ROT(n1+k2+π+n3, n1)+k1+n3, n2)+n3
n1' = MIXBITS(n3, n2);
C = ROT((ROT(n3+k1*+π+n1', n3)+k2*⊕n1', n2)⊕n1'
2.1. "A||B||C"
2.2. "D"
D' = ROT((ROT(n2+k2*+ID+n1',n2)+k1*+n1', n3)+n1'
If D' = D
3. Back-end database updating
Extracts n1 from A, and n2 from B.
n3=MIXBITS(n1, n2)
k1*=ROT((ROT(n2+k1+π+n3, n2)+k2⊕n3, n1)⊕n3
k2*=ROT((ROT(n1+k2+π+n3, n1)+k1+n3, n2)+n3
n1' = MIXBITS(n3, n2)
C'= ROT((ROT(n3+k1*+π+n1', n3)+k2*⊕n1', n2)⊕n1'
If C' = C
D = ROT((ROT(n2+k2*+ID+n1',n2)+k1*+n1', n3)+n1'
3. Tag updating
n2' = MIXBITS(n1', n3)
IDSold = IDS; IDSnext= ROT((ROT(n1'+k1*+IDS+n2', n1')+k2*⊕n2', n3)⊕n2'
k1old = k1; k1next= ROT((ROT(n3+k2*+π+n2', n3)+k1*+n2', n1')+n2'
k2old = k2; k2next= ROT((ROT(IDSnext+k2*+π+k1next, IDSnext)+k1*+k1next, n2')+k1next
IDS = ROT((ROT(n1'+k1*+IDS+n2', n1')+k2*⊕n2', n3)⊕n2'
k1= ROT((ROT(n3+k2*+π+n2', n3)+k1*+n2', n1')+n2'
k2= ROT((ROT(IDS+k2*+π+k1, IDS)+k1*+k1, n2')+k1
π = 0x3243F6A8885A308D313198A2 (L = 96 bits)

Fig. 1 Gossamer protocol specification

a new N-bit number. Since the function has to be executed also on the tag, it uses only the operations allowed on the tag.

Figure 1 provides an overall description of the protocol. Notice that π is a 96-bit string, in hexadecimal notation, equal to $0x3243F6A8885A308D313198A2$. The reader starts by sending a "Hello" message to the tag. The tag replays with its current IDS. If something went wrong in the previous interaction, the reader does not recognize this current pseudonym, and sends again a "Hello" message. Then, the tag replays with the old IDS.

When the reader recognize the IDS and, thus, can retrieve the keys, k_1 and k_2, associated to the tag, generates two random numbers, n_1 and n_2, and computes the N-bit values, A and B, as

$$A = \mathsf{Rot}((\mathsf{Rot}(IDS + k_1 + \pi + n_1, k_2) + k_1, k_1),$$
$$B = \mathsf{Rot}((\mathsf{Rot}(IDS + k_2 + \pi + n_2, k_1) + k_2, k_2).$$

Then, it uses the $\mathsf{MixBits}$ function to obtain an N-bit string $n_3 = \mathsf{MixBits}(n_1, n_2)$. New keys k_1^* and k_2^* are, afterwards, generated by performing left circular rotations operations. Precisely:

$$k_1^* = \mathsf{Rot}(\mathsf{Rot}(n_2 + k_1 + \pi + n_3, n_2) + k_2 \oplus n_3, n_1) \oplus n_3,$$
$$k_2^* = \mathsf{Rot}(\mathsf{Rot}(n_1 + k_2 + \pi + n_3, n_1) + k_1 + n_3, n_2) + n_3$$

Finally, the function $\mathsf{MixBits}$ is used again to obtain another N-bit string $n_1' = \mathsf{MixBits}(n_3, n_2)$ and, in turn, this is used, together with n_2, n_3 and the new keys k_1^* and k_2^*, to produce the value

$$C = \mathsf{Rot}(\mathsf{Rot}(n_3 + k_1^* + \pi + n_1', n_3) + k_2^* \oplus n_1', n_2) \oplus n_1'.$$

At this point, denoting with $||$ the string concatenation operator, the message $A||B||C$ is sent by the reader to the tag.

Upon reception of the message $A||B||C$, the tag extracts n_1 from A and n_2 from B. The tag can do this because it knows k_1, k_2 and IDS. Hence, it can reverse the computation, made by the reader, to compute A (resp. B) from k_1, k_2, IDS and n_1 (resp. and n_2). Once retrieved n_1 and n_2, the tag performs exactly the same computation that the reader has performed

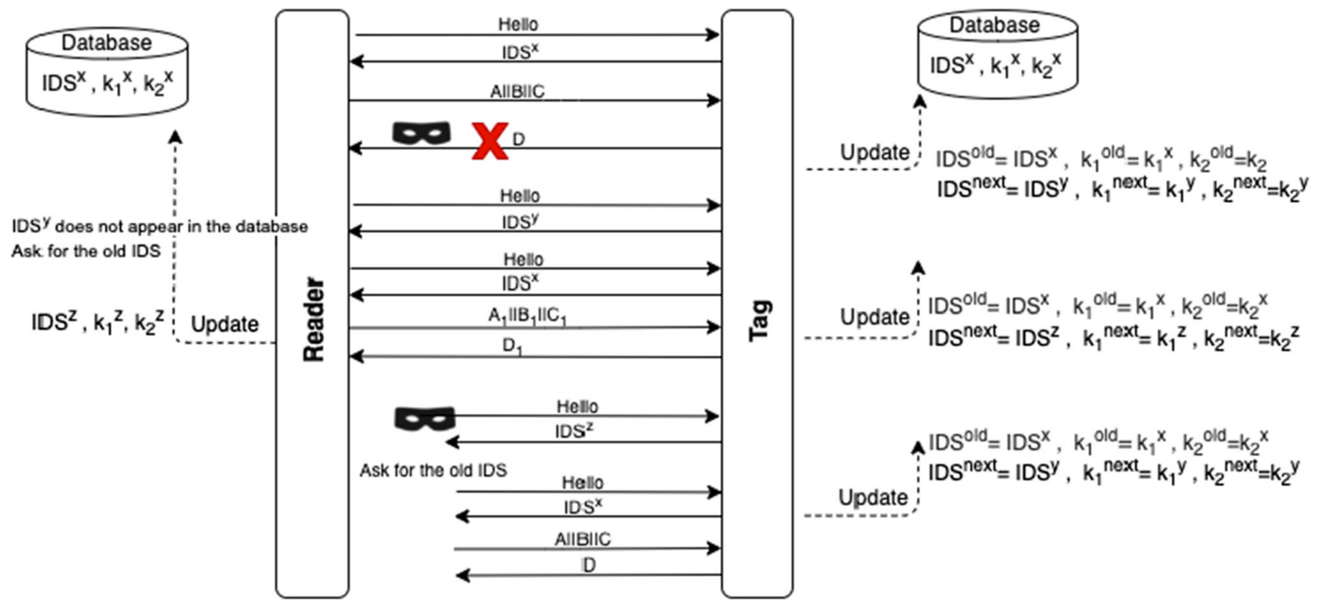
Database
IDS^x, k1^x, k2^x
Hello
IDS^x
A||B||C
D
Hello
IDS^y
IDS^y does not appear in the database
Ask for the old IDS
Hello
IDS^x
A1||B1||C1
D1
IDS^z, k1^z, k2^z
Update
Reader
Tag
Hello
IDS^z
Ask for the old IDS
Hello
IDS^x
A||B||C
D
Database
IDS^x, k1^x, k2^x
Update
IDS^old= IDS^x, k1^old= k1^x, k2^old=k2
IDS^next= IDS^y, k1^next= k1^y, k2^next=k2^y
Update
IDS^old= IDS^x, k1^old= k1^x, k2^old=k2^x
IDS^next= IDS^z, k1^next= k1^z, k2^next=k2^z
IDS^old= IDS^x, k1^old= k1^x, k2^old=k2^x
IDS^next= IDS^y, k1^next= k1^y, k2^next=k2^y
Update

Fig. 2 A de-synchronization attack on Gossamer

to obtain C: that is, it computes $n_3 = \mathsf{MixBits}(n_1, n_2)$ and the two new keys k_1^* and k_2^*, and, finally, the value C'. We have written C' because this is the tag's "local" version of C. At this point, the tag can check whether $C' = C$, to validate the entire message. If this check is successful, then the tag performs the updating of the keys and of the pseudonym, and responds to the reader with the value

$$D = \mathsf{Rot}(rot(n_2 + k_2^* + ID + n_1', n_2) + k_1^* + n_1', n_3) + n_1'.$$

Upon reception of the message D, the reader can compute D', its "local" version of D and, finally, check whether $D' = D$. If the check is successful, the authentication has succeeded also for the reader, and the reader too performs the updating of the keys and of the pseudonym.

2.3 Known attacks

Some vulnerabilities of Gossamer, exploited by passive and active attacks, are known. Regarding the first ones, two passive attacks have been reported in [3]. However, these attacks require that nonces and key values be zero. Indeed, the rotation and MixBits functions depend on key and nonce values. If these values are zero, the outputs are not pseudorandom. For these special cases, thus, are carried out a full disclosure and an ID disclosure attacks on the protocol.

Ahmed et al. [3] suggest to avoid these vulnerabilities by modifying the MixBits function, to ensure that its output is not zero, even when its two inputs are equal to zero.

Concerning the second one, the security analysis in [9] shows that Gossamer is vulnerable to a de-synchronization attack [40]. The attack is reported in Fig. 2.

An attacker first eavesdrops the authentication messages exchanged between a reader and a tag, i.e., the message $A||B||C$. Then, it blocks the last message, sent to the reader by the tag, i.e., the message D, preventing the updating of the shared secrets, i.e., (IDS^x, k_1^x, k_2^x) in the reader database. Afterwards, the attacker waits for the reader to initiate and complete successfully a genuine authentication session with the tag. Notice that, the tag, during such a new session, sends first the new IDS^y, but the reader does not reply because the reader has not updated his database. Thus, the tag sends out its old IDS^x. Later on, the attacker initiates an authentication session with the tag. When it receives the new IDS^z, it does not reply, waiting to get the old one, IDS^x. The eavesdropped message $A||B||C$, from the first session, is then replayed to the tag. Such a message trigger the tag to update its database and to be, as Fig. 2 clearly shows, finally, de-synchronized with the reader.

A simple solution for avoiding the de-synchronization attack is presented in the same paper. It is suggested that the reader sends to the tag a message that uses the new shared secrets before the tag updates the shared secrets. One year later, Gossamer was revised again in [39], to prevent the de-synchronization attack, by storing both the old and the new shared secrets on the tag and on the reader sides. Later on, in [42], the authors introduced an improved key-update mechanism, based on a lightweight pseudonym random generator, that always extracts the new secrets from the old ones, rather than creating a whole new value at the end of each session. The newly updated secrets on the tag and the reader database will always be the same, even if the attacker attempts to cause a de-synchronization attack.

3 Classification problems, machine learning, and deep learning

In this section, we briefly describe well-known AI techniques, that we will exploit in subsequent sections, to build distinguishers for MixBits outputs and for messages of the Gossamer protocol. We will cast the problem as a classification question and we will build adhoc classifiers. A *classification* problem consists of identifying the category to which an object belongs. In our case, the objects are strings of bits representing random strings, MixBits outputs, and Gossamer messages. An algorithm capable of classifying the objects is a *classifier*.

As a first attempt, we took into consideration common machine learning techniques, namely, Decision Tree (DT), Naive Bayes (NB), shallow Neural Network (NN), k-Nearest Neighbors (k-NN), and Support Vector Machine (SVM) to build a classifier. Then, in order to achieve a better classifier, we looked at more sophisticated machine learning models, that is, deep learning techniques. More precisely, we turned to consider Siamese networks. We emphasize that such networks are designed to tackle a similarity problem, not a classification one, but it is quite easy to turn a binary classification problem into a similarity problem; and the classification problem we study is a binary classification.

3.1 Supervised learning

A classifier is first trained on known objects and, then, used to classify unknown objects. The classification may be supervised or unsupervised, depending on whether the objects are labeled or unlabeled. A label on an object specifies the class of the object; for example, a 96-bit string can be labeled as a "MixBits output." A supervised classifier is trained on labeled objects, and exploits a so-called *feature vector*, which is derived from the objects themselves. After the training, the trained classifier is tested on a reasonable number of previously unseen objects. We will use supervised classifiers.

3.2 Rapidminer and standard machine learning

To implement a classifier, one can use public libraries available for many programming languages. There exist also easy to use tools that simplify the implementation, since no coding is necessary. Rapidminer[1] is a user-friendly platform for machine learning and data mining applications, that allows to fine tune specific parameters. The reader can find more information and the documentation in [1,28]. For the standard approaches, we used Rapidminer to build the classifiers.

To evaluate the performance of the classifiers, Rapidminer allows to use the k-fold cross-validation method. We used it choosing $k = 10$, i.e., 10-*fold cross-validation* (CV for short). The 10-fold CV randomly divides the dataset into 10 equal-sized parts. A single part out of the 10 is held as validation set for evaluating the classifier, while the other 9 parts are used as training set. The cross-validation process is then repeated 10 times, with each of the 10 parts serving as validation set exactly once. Results are collected from the 10 executions and averaged to produce an overall performance estimation.

Rapidminer lets us choose a variety of standard evaluation metrics. We have selected the most common ones: accuracy, precision, and recall. In a binary classification problem, the objects from the two classes are normally labeled as positives and negatives. Accuracy is computed as the number of correctly classified objects, belonging to both categories, divided by the total number of classified objects. Precision is computed as the number of correctly classified objects as positive, out of all those classified positive. Recall is computed as the number of correctly classified objects as positive, divided by the total number of positive objects.

More precisely, letting TP be the number of true positives, TN be the number of true negatives, FP be the number of false positives, and FN be the number of false negatives, the three metrics are defined as:

$$\text{Accuracy (ACC)} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{1}$$

$$\text{Precision (PRC)} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2}$$

$$\text{Recall (RCL)} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{3}$$

3.3 Classifiers based on deep learning

In order to improve the results that we obtained with standard classifiers, we explored also alternative models, namely, Siamese neural networks. A Siamese network consists of two identical sub-networks, run side-by-side, each of which takes as input an object. The output of the entire Siamese network is a *similarity measure* of the two inputs.

Siamese networks have proved to be a powerful tool for tackling several problems. For example, in 1994, a Siamese neural network was used to detect forged signatures [10]. Precisely, the network was able to determine, by comparing them, whether two handwritten signatures were original or one of them was a forgery. Since then, Siamese neural networks have been used for various applications [12,27]. For further details, the interested reader is referred to [27].

Siamese networks are based on the so-called one-shot learning. One-shot learning classifies objects given only one training example for each category of objects. Indeed, instead of directly classifying objects and assigning them to the categories for each test object, a Siamese network takes an extra

[1] `www.rapidminer.com`.

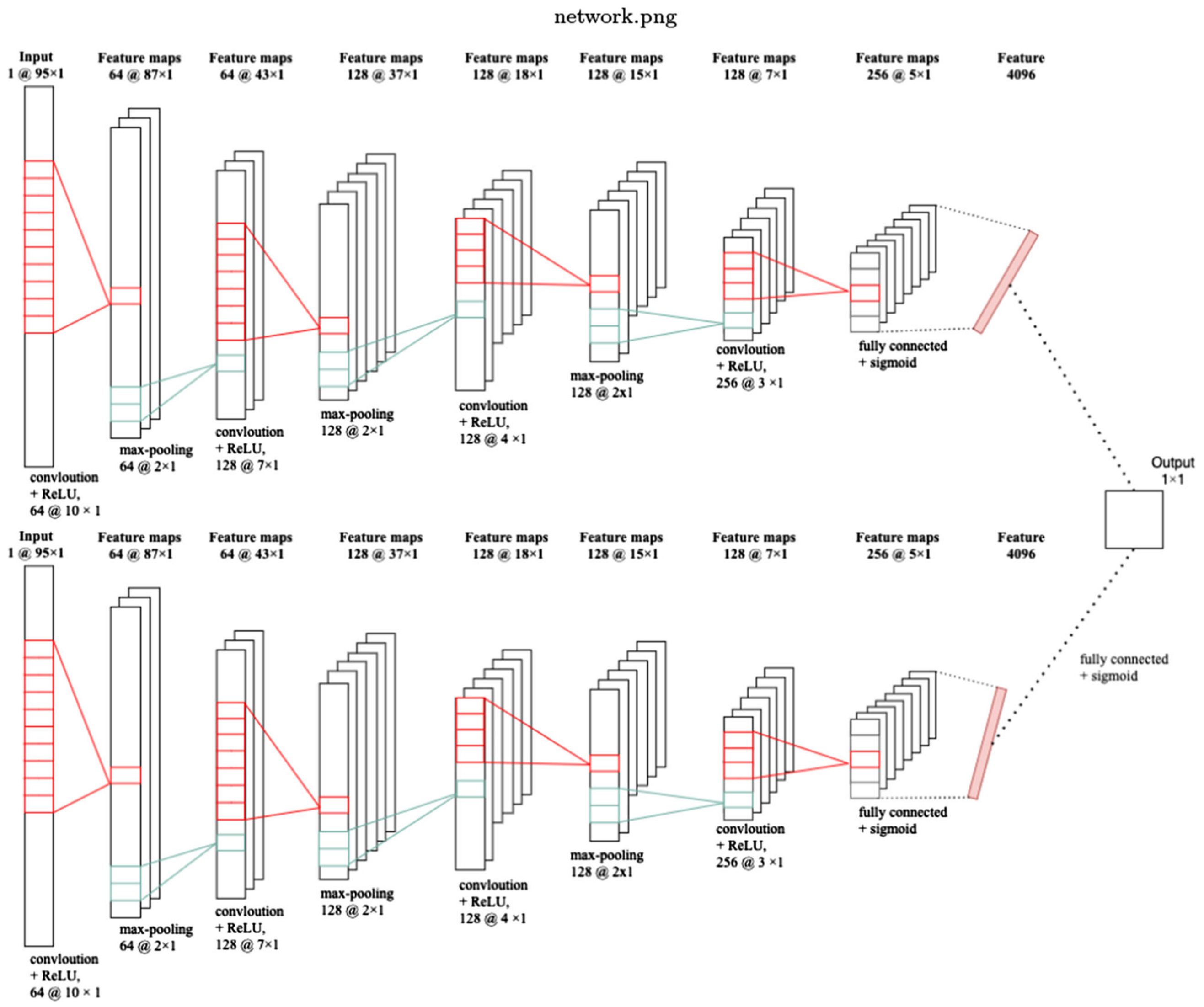

Fig. 3 Convolutional Siamese neural network architecture using the 95-bit feature vector

reference object as input and produces a similarity score, denoting the chances that the two input objects belong to the same category. Typically the similarity score is a value between 0 to 1, where score 0 denotes no similarity, and score 1 denotes full similarity.

After several attempts, with various configurations of the network, we ended up implementing and using a Siamese network based on convolutional neural networks. A convolutional neural network is a particular kind of neural network, specifically designed for dealing with images, but that has proved to be effective also for several other problems. Our convolutional Siamese network is depicted[2] in Fig. 3.

[2] For readers interested in the details and with background in the area, each sub-network consists of a sequence of convolutional layers, each of which uses a single channel, with filters and kernels of varying size. The number of convolutional filters is specified as a multiple of 16, to get advantage of the GPU capabilities. The network applies a ReLU activation function to the output feature maps, optionally followed by a one-dimensional max-pooling layer. The units in the final convolutional layer are flattened into a single vector. This convolutional layer is followed by a fully-connected layer and, then, one more layer, computing the induced distance metric between each siamese sub-network, which is given to a single sigmoidal output unit.

4 An in-depth look at the MixBits function

The Gossamer protocol uses an ultra-lightweight function, called MixBits. The MixBits function takes as input two N-bit numbers, and produces an N-bit number, by using only right shifts and additions. In Gossamer, MixBits is used with $N = 96$. Denoting with $Z >> a$ denotes the right shift of Z of a positions, with $a \geq 0$, the function is the following.

```
MixBits($X$, $Y$)
  $Z = X$;
  for ($i = 0$; $i < 32$; $i$++)
    $Z = (Z >> 1) + Z + Z + Y$;
  return $Z$
```

The MixBits function has been defined in [33]; the technique used to design the function has been proposed in [20] and is quite interesting: the idea is to use genetic programming to evolve a population of ultralightweight functions, starting from very basic bitwise operations, like `rotations`, `and`, `or`, `xor`, `not` and `sum`. The genetic search is guided by a fitness function that measures the *avalanche* effect, which is useful to assess the nonlinearity of the functions.

Let x and y be two N-bit strings, and denote with $H(x, y)$ their Hamming distance, i.e., the number of positions in which x and y are different. A function $f : \{0, 1\}^N \to \{0, 1\}^N$ exhibits the avalanche effect if,

$$\forall x, y \text{ such that } H(x, y) = 1,$$

it holds that on average $H(f(x), f(y)) = N/2$.

In other words, f has the avalanche effect if, flipping one bit in the input, causes the output to change, on average, in half of the bits (see, for example, [25]). The genetic algorithm used to find MixBits looked for the function with the best avalanche effect.

4.1 The avalanche effect of MixBits

To analyze the avalanche effect of MixBits we performed the following test many times: take two random 96-bit numbers, x and y, and compute $z =$ MixBits(x, y); then, flip a random bit of x, to obtain x', and compute $z' =$ MixBits(x', y); measure the Hamming distance between z and z'. More precisely, we used Algorithm 1.

```
1 Choose two random numbers $x, y \in \{0, 1\}^{96}$;
2 Compute $z$ = MixBits$(x, y)$;
3 $i \leftarrow$ random(0, 95);
4 $x' = x$ with bit number $i$ flipped;
5 Compute $z'$ = MixBits$(x', y)$;
6 Output $H(z, z')$;
```

Algorithm 1: MixBits avalanche effect test

The avalanche effect should produce an average difference between z and z' of 48 bits; in general, we would expect the distribution of $H(z, z')$ to be a normal distribution centered at 48. The test was repeated 100,000 times, and Fig. 4 shows both the expected normal distribution (blue vertical columns) and the MixBits distribution (green line), obtained in the test.

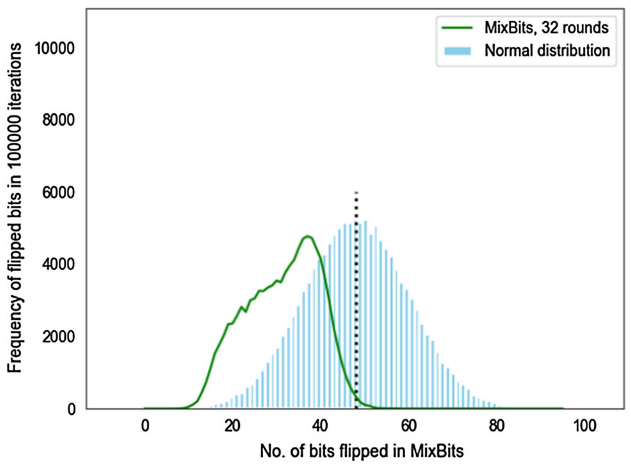

Fig. 4 MixBits avalanche effect

The x-axis reports the number of bits that have been flipped, that is, $H(z, z')$, while the y-axis reports the number of times that each specific value of $H(z, z')$ has been given as output. As can be seen from the figure (green line), the distribution is somewhat similar to a normal one, but is centered around 38, instead of 48.

MixBits, as designed in [33], uses 32 rounds; in that paper, there is no explicit justification for this choice, neither the MixBits function is explicitly mentioned in [20]. We, thus, have tried the test using all possible choices for the number of rounds, that is from 1 to 96. Not surprisingly, the avalanche effect improves with the number of rounds. Denoting with MixBits(r) the MixBits function with r rounds, we have that the center of the distribution of the avalanche effect of MixBits(r) goes from close to 0, for small values of r, to 48, for values of r approaching 96. Moreover, the shape of the distribution goes from a "distorted" bell-shape, for small values of r, to an almost perfect bell-shape, for big values of r.

Figure 5 shows the avalanche effect of MixBits as a function of the number of rounds for a few selected values of the number of rounds, namely the ones in the set $\{1, 16, 24, 32, 48, 60, 72, 96\}$. Using a large number of rounds, e.g. bigger than 70, yields an almost perfect avalanche effect; with few rounds, the center of the distribution is very displaced, and the shape of the distribution is somewhat distorted. Based on these observations, it seems that 32 rounds is a choice that provides a good balance between the computational power required to compute the function and the avalanche effect achieved.

Notice that, from a theoretical perspective, MixBits can be seen as a keyed function $F(\cdot, \cdot)$ which, for each choice of the second argument, the *key*, defines a single-argument mapping, i.e., $F(\cdot, y) = F_y(\cdot)$. In different but equivalent terms, it defines a family of functions $\{F_y(\cdot)\}_y$.

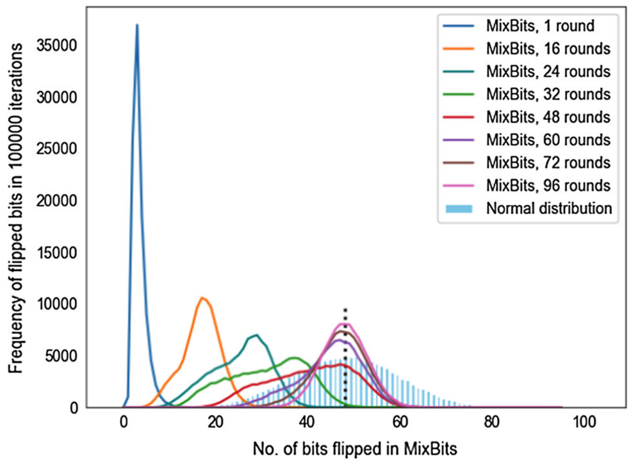

Fig. 5 MixBits avalanche effect distribution with several values for the number of rounds

An important property keyed functions might enjoy is *pseudorandomness*: roughly speaking, to the eyes of any efficient observer, they behave like random functions, i.e., domain values and range values seem to the observer to be totally unrelated (see for example [25]). Pseudorandomness of $F(\cdot,\cdot)$ can be defined using the following Pseudo-R experiment, defined for any algorithm D:

Pseudo-R$_{D,F}$

1. A bit b is chosen. If $b = 1$, then a key $y^* \in \{0,1\}^{96}$ is chosen, uniformly at random, and an oracle $\mathcal{O}(\cdot)$ is set to reply to queries using $F(\cdot, y^*)$. Otherwise, a function $f : \{0,1\}^{96} \rightarrow \{0,1\}^{96}$ is chosen, uniformly at random, from the set of all the functions of 96 bits to 96 bits, and $\mathcal{O}(\cdot)$ is set to reply to queries using $f(\cdot)$.
2. D receives access to oracle $\mathcal{O}(\cdot)$, and gets replies to at most t queries.
3. D outputs a bit b'.
4. The output of the experiment is 1 if $b' = b$, 0 otherwise.

The keyed function F is (t,ϵ)-pseudorandom if, for any algorithm D (the *distinguisher*), which runs for at most t steps, there exists a small ϵ, such that

$$Pr[\text{Pseudo-R}_{D,F} = 1] \leq 1/2 + \epsilon.$$

The above analysis enables us to set up simple distinguishers for MixBits: one possibility could be for the distinguisher, with many trials, to estimate the probability distribution of the replies from the oracle, i.e., to check whether the output differences follow the green curve (MixBits) or the normal distribution (truly random function). Actually, to prove our claim, a simpler strategy is enough. Look at the distinguisher $\mathcal{D}$, given by Algorithm 2.

```
1 Chooses $x$ and $x'$, where $x'$ is equal to $x$ up to one bit;
2 Sends $x$ to the oracle, getting $z$ as reply;
3 Sends $x'$ to the oracle, getting $z'$ as reply;
4 Computes the integer $d = H(z, z')$;
5 If $d < 48$, then outputs 0 (MixBits); otherwise, outputs 1 (truly
  random);
```

Algorithm 2: Code for the distinguisher $\mathcal{D}$.

Due to the shapes of the curves in Fig. 4, for the MixBits function, the $Pr(d < 48) \approx 1$. While, for a truly random function, the $Pr(d < 48) < 1/2$. Since the oracle implements MixBits or a truly random function, depending on the choice, uniformly at random, of a bit b, it holds that:

$$\begin{aligned} Pr(\text{Pseudo-R}_{D,F} = 1) &= Pr(b=0)\cdot Pr(d < 48|b=0) + \\ &+ Pr(b=1)\cdot Pr(d \geq 48|b=1) \\ &\simeq \frac{1}{2}\cdot 1 + \frac{1}{2}\cdot[1 - Pr(d < 48|b=1)] \\ &\geq \frac{1}{2}\cdot 1 + \frac{1}{2}\cdot\left(1 - \frac{1}{2}\right) \\ &= \frac{1}{2} + \frac{1}{2}\cdot\frac{1}{2} = \frac{1}{2} + \frac{1}{4} = \frac{3}{4} \end{aligned}$$

It follows that, with advantage at least of $\frac{1}{4}$ over a random guess, the distinguisher $\mathcal{D}$, with only two queries, is successful. Hence, MixBits does not realize a pseudorandom function.

4.2 A generalization of MixBits

Informally, the MixBits function could be described as the function $3x/2 + y$, repeated 32 times, with the value of x for the next iteration divided by 2. So, informally, we can look at MixBits as a special case of the following function,

$$f(x, y) = a \cdot x/2^c + b \cdot y/2^d$$

with parameters $a \geq 1, b \geq 1, c \geq 0$ and $d \geq 0$, and the value of x and y updated to, respectively, $x/2^c$ and $y/2^d$. An additional parameter r is the number of rounds. More formally, we define the following GenMixBits function, parametrized on a, b, c, d and r:

```
GenMixBits($X, Y$)
  $Z = X$;
  $W = Y$;
  for ($i = 0$; $i < r$; $i$++)
    $Z = (Z >> c) + \underbrace{Z + Z + \ldots + Z}_{a-1 \text{ times}} +$
        $(W >> d) + \underbrace{W + W + \ldots + W}_{b-1 \text{ times}}$;
  return $Z$
```

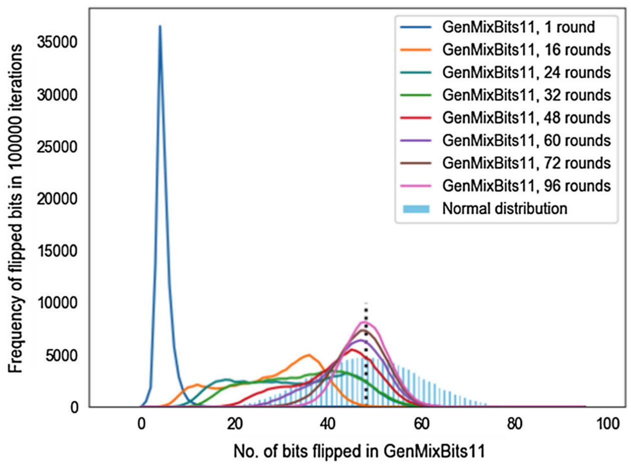

Fig. 6 GenMixBits11 $11x/2 + y$, with several values for the number of rounds

Thus, MixBits is the special case of GenMixBits with $a = 3, b = 1, c = 1, d = 0$ and $r = 32$.

We studied the avalanche effect of GenMixBits for several choices of the parameters: we considered all combinations of *(relatively) small* values of a, b, c, d, and r. Among these, we found another interesting case in which we obtain a good avalanche effect. Namely, the case with parameters $a = 11, b = 1, c = 1, d = 0$, which informally is the function $11x/2 + y$. We will refer to this particular choice of the parameters as the function GenMixBits11.

As done for MixBits we have considered all possible values for the number r of rounds. Figure 6 shows the avalanche effect of GenMixBits11 for several choices of r. For any fixed value of r, the avalanche effect of this function seems to be better than that of MixBits. This is clearly due to the factor of 3 being replaced with 11, which obviously requires more computation; however, GenMixBits11, with only 16 rounds, shows an avalanche effect similar to that of MixBits with 32 rounds.

The above analysis suggests us *possible improvements* for future design strategies: indeed, MixBits was found by using genetic programming, evolving a population of ultralightweight functions, and selecting one at the end. Traditional statistical tests, like the one we presented earlier, could be used to refine the analysis of the quality of the selected functions and, as the example of GenMixBits shows, once the *form* of the function has been found, it is possible to look for similar other instances with post-processing methods.

4.3 On the distinguishability of MixBits outputs through AI techniques

In this section, we study how well the MixBits function approximates at least a *weak* pseudorandom function. Ideally, it should not be possible to tell whether a given string has been given as output by MixBits or it is a random string, when the input strings are chosen *uniformly at random*. That is, compared to the previous case, a distinguisher cannot use x and x', where x' is equal to x up to one bit (or few bits) for example, but only queries where x and x' are chosen *uniformly at random*. In such a setting, the output of MixBits should be indistinguishable from the output of a truly random function. We show that this is not the case, too. We produce both random numbers and MixBits outputs, and check whether we can tell them apart. We first focus on the study of the MixBits function using machine learning classifiers. Then, to improve the result obtained, we consider deep learning techniques, exploiting also an alternative representation (feature vector) of the objects. In our context, the objects are 96-bit strings and belong to one of two categories: random strings or MixBits outputs.

Datasets We have considered two cases, the first one in which we evaluated the weak pseudorandomness of MixBits, and the second one in which we evaluated the difficulty of distinguishing the outputs of *different instances* of MixBits from random values. The latter problem is more difficult since the key y, used by the MixBits function, is changed in each new evaluation. We will refer to the former case as the *weak* case, and the latter as the *strong* case. For each case, we performed an experiment, repeated 100 times. For each repetition of the experiment, a dataset of 3000 objects has been built as follows.

Datasets for the weak case. Each dataset is the union of two sets. The first one is a set of 1500 random 96-bit numbers. The second one is a set of 1500 MixBits(x, y) outputs, obtained by fixing a randomly chosen y and, then, executing MixBits on 1500 randomly chosen values of x (and with the fixed y).

Datasets for the strong case. As for the weak case, but with the difference that the sets of MixBits(x, y) output where produced by choosing *both* x and y at random. More precisely, for each experiment, we built a set of 1500 random 96-bit numbers, and a set of 1500 MixBits(x, y) outputs, obtained by choosing, for 1500 times, two random values for x and y. The MixBits function has been implemented using the GMP library from GNU; all the random values that we describe in the following have been generated by using the pseudorandom number generator of the GMP library.

4.3.1 Experimental results

Standard classifiers With the classifiers implemented through Rapidminer, we obtained the following results: an accuracy of 0.5, a precision of 0.5 and a recall of 0.584 using k-NN for the MixBits strong case, and an accuracy of 0.527, a precision of 0.525 and a recall of 0.546, using DT for the MixBits weak case. Table 1 describes the configurations for the parameters

Table 1 Configurations of the classifiers

Classifier	Parameter configurations
DT (C4.5)	Maximum depth = 7, criterion = Gini index, pruning = True, confidence = 0.25
k-NN	k = 25, weighted vote = True

that we have chosen as a result of several attempts to find the best rates.

Such values for accuracy, precision, and recall mean that these standard classifiers are not quite successful for the problem we are considering.

Siamese networks As anticipated, a Siamese network, by definition, solves a similarity problem. However, a binary classification problem can easily be turned into a similarity problem. In our case, we need to distinguish whether a bitstring s is random or is a MixBits output. To cast this problem into a similarity problem, we can ask whether s is similar to a random bit string s_r (that we can build) or to a MixBits output s_m (that we can compute).

To use the Siamese network described in Sect. 3 for the MixBits indistinguishability experiment, we have built a dataset, consisting of two subsets: one including pairs of similar bitstrings, and one including pairs of not similar bitstrings. To this end, starting from one of the datasets used before (randomly chosen), consisting of 1500 bitstrings from MixBits, and of 1500 truly random bitstrings, we have built all the possible pairs, and, then, we have split them in two sets:

- *similar-set*, including all the pairs $< s_i, s_j >$, such that both are outputs of MixBits, or both are outputs of a truly random function,
- *non-similar-set*, including all the pairs $< s_i, s_j >$, such that s_i (resp. s_j) is output of MixBits, and s_j (resp. s_i) is output of a truly random function.

As *training-set* for the Siamese network, we have randomly chosen 5000 pairs from *similar-set*, and 5000 pairs from *non-similar-set*. We have trained the network for 2000 epochs, using a learning rate of 0.00006, and using the 100-way one-shot learning technique. Specifically, every 5 epochs (one-shot evaluation interval), our model first chooses an anchor object $\widehat{s}$, and then builds 100 pairs $< s'_i, s''_i >$ with $i = 1, \ldots, 100$, where:

- s'_i and s''_i are randomly chosen in *similar-set* $\cup$ *non-similar-set* (resp. $\{$*similar-set* $\cup$ *non-similar-set*$\}$ $\setminus$ *training-set*), for $i = 2, \ldots, 100$.
- $s''_1 = \widehat{s}$, and s'_1 belong to the same class, so that $s'_1, s''_1 >$ is the only one pair of objects in the same class, among the 100 pairs.

The set of 100 pairs obtained is used as validation set (resp. test set). In other words, the same object is compared to 100 different objects, out of which only one of them matches the original object. For every pair in the validation set (resp. test set), the network generates a similarity score between 0 and 1. Let us say that, by doing the above 100 comparisons, we get 100 similarity scores $S_1, \ldots, S_{100}$. Now, if the model is trained properly, we expect that S_1 is the maximum of all the 100 similarity scores, because the first pair is the only one where we have two objects in the same category. Thus, if S_1 happens to be the maximum score, then we treat this as a correct prediction; otherwise, we consider it as an incorrect prediction. Repeating this procedure ℓ times, both for the validation set and the test set, we compute the percentage of correct predictions as

$$pcorrect = (100 * ncorrect)/\ell,$$

where ℓ is the total number of trials (in our case is 2000/5, where 5 is the one-shot evaluation interval), and *ncorrect* is the accuracy out of ℓ trials.

Results We have repeated the experiment 10 times and, as a result, we have obtained an average test accuracy of 0.56 for the MixBits strong case, and of 0.586 for the MixBits weak case, as reported in Table 2.

4.3.2 An alternative feature vector

As the dataset consists of rows of data of 96-bit numbers, the immediate choice for the features to be used for the classification is that of using directly the 96 bits; that is, feature number i is the ith bit. Thus, we initially used such a representation, obtaining the results reported in the previous subsection. However, in order to get the best possible performance from the Siamese neural network, we tried other possibilities and ended up using the following 95-number features vector: for $i = 1, \ldots, 95$, feature number i is the difference, in absolute value, of the number of 1s and the number of 0s in the first $i + 1$ bits. This particular feature vector conveys information about how balanced is each pre-

Table 2 MixBits prediction accuracy of the Siamese neural network (S = Strong case, W = Weak case)

	1	2	3	4	5	6	7	8	9	10	Avg
S	0.54	0.55	0.58	0.57	0.57	0.55	0.56	0.56	0.54	0.58	0.560
W	0.56	0.61	0.57	0.57	0.59	0.59	0.58	0.6	0.59	0.6	0.586

Table 3 MixBits prediction accuracy of the Siamese neural network, using the 95-bit feature vector (S = Strong case, W = Weak case)

	1	2	3	4	5	6	7	8	9	10	Avg
S	0.64	0.67	0.67	0.71	0.72	0.69	0.69	0.7	0.68	0.71	0.688
W	0.66	0.68	0.72	0.7	0.75	0.72	0.72	0.73	0.74	0.74	0.716

fix of the 96-bit string and the experimental data show that this information helps the learning process. As an example, consider the 10-bit data string 1011101110; the 9-number feature vector would be 012323454.

We have repeated the experiment of Sect. 4.3.1 for 10 times, with the 95-bit feature vector, and we have obtained an average accuracy of 0.688 for the MixBits strong case, and of 0.716 for the MixBits weak case, as reported in Table 3.

4.4 Impact of the weak avalanche effect of MixBits on Gossamer

In the previous section, we checked the distinguishability of MixBits outputs and random strings, concluding that they are distinguishable. Since MixBits is used in Gossamer, we found interesting to test also the avalanche effect on the overall Gossamer protocol. We studied the differences that a bit flipped in message A of Gossamer causes in message C. Since A is obtained from n_1 and the two keys k_1, k_2, message A' can be thought of as the message obtained from $n'_1 \neq n_1$ and the two keys k_1, k_2. By inverting the computation used for message A we can compute n'_1. More formally, we consider the test detailed by Algorithm 3.

```
1 Execute the reader authentication phase, that is compute $A, B, C$
2 Flip a random bit of $A$ obtaining $A'$
3 Compute
  $n'_1 = \mathsf{Rot}(\mathsf{Rot}(A', 96 - k_1) - k_1, 96 - k_2) - IDS - k_1 - \pi$
4 Compute $n'_3 = \mathsf{MixBits}(n'_1, n_2)$
5 Compute
  $k_1^{*'} = \mathsf{Rot}(\mathsf{Rot}(n_2 + k_1 + \pi + n'_3, n_2) + k_2 \oplus n'_3, n'_1) \oplus n'_3$
6 Compute
  $k_2^{*'} = \mathsf{Rot}(\mathsf{Rot}(n'_1 + k_2 + \pi + n'_3, n'_1) + k_1 + n'_3, n_2) + n'_3$
7 Compute $n''_1 = \mathsf{MixBits}(n'_3, n_2)$
8 Compute
  $C' = \mathsf{Rot}(\mathsf{Rot}(n'_3 + k_1^{*'} + \pi + n''_1, n'_3) + k_2^{*'} \oplus n''_1, n_2) \oplus n''_1$
9 Output $H(C, C')$.
```

Algorithm 3: Gossamer avalanche effect test

The test has been repeated 100,000 times and the results are summarized in Fig. 7. The x axis shows the number of flipped bits, and the y axis shows the percentage of experiments that gave the specific avalanche effect. As it can be seen from the figure, the distribution is very similar to a normal distribution centered in 48.

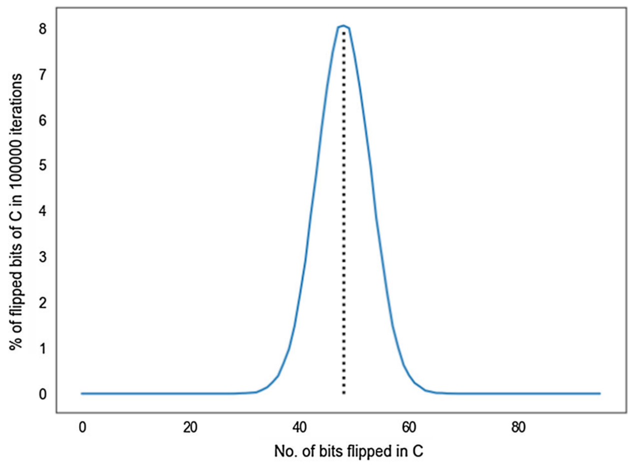

Fig. 7 Gossamer avalanche effect

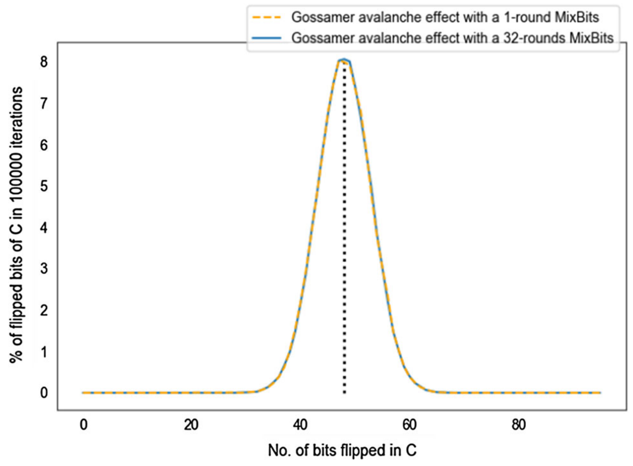

Fig. 8 Gossamer avalanche effect

Since MixBits does not have a good avalanche effect, this means that the other operations of the protocol compensate for the weaknesses of MixBits.

Another observation that goes in this direction is the following: the avalanche effect of the entire protocol *does not* depend on the number of rounds of MixBits. To prove this statement, we carried out the following experiment: we flipped one single (randomly chosen) bit of A, and looked at the avalanche effect for the standard MixBits function, and a simplified MixBits function that uses only one round. This experiment was carried out 100,000 times and there were no noticeable differences. Figure 8 reports the details of the experiment: the blue and orange lines show the avalanche

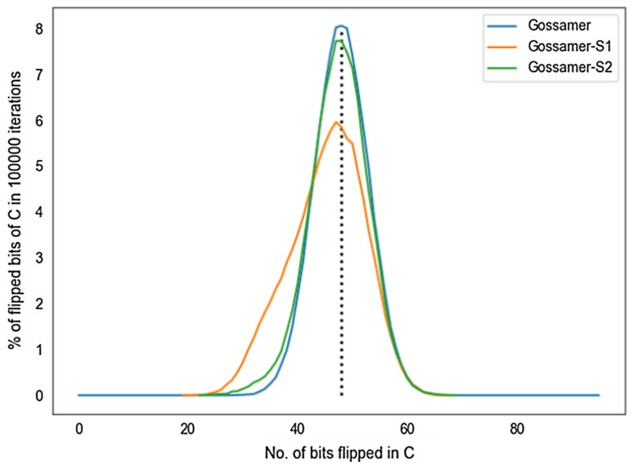

Fig. 9 Comparison of the Gossamer and its simplified versions for avalanche effect

effect on C for one round, and the thirty-two rounds versions of MixBits: they are essentially the same.

4.5 Avalanche effect analysis

As suggested by the results of the previous sections, the avalanche effect on C is mostly due to operations of the Gossamer protocol, with a mild contribution from the MixBits function. So we decided to study what are the contributions of some of the protocol operations to the avalanche effect. To this aim, we considered two simplified versions of the protocol, which we built having Gossamer as a guide, but using simplified computations for the construction of the messages. More precisely, in the first version, called Gossamer-$S1$, we use only the $+$ operation while, in the second version, called Gossamer-$S2$ one rotation is added. The code is reported in Figs. 11 and 12 at the end of the paper. In both versions, the MixBits function is used without changes.

We repeated the test performed in Sect. 4.4: check the avalanche effect when flipping a bit of message A, as described in Algorithm 3. As done for Gossamer, the test was repeated 100,000 for Gossamer-$S1$ and for Gossamer-$S2$.

Figure 9 shows the results of the tests and compares the simplified versions of the protocol with the original one. The x-axis shows the number of flipped bits, and the y-axis shows the percentage of experiments that gave the specific avalanche effect. The orange line shows the avalanche effect of Gossamer-$S1$, the green line shows the avalanche effect of Gossamer-$S2$, while the blue line shows the avalanche effect of the Gossamer protocol. The avalanche effect is more or less the same. In conclusion, putting everything together, it is interesting to notice that, if we reduce the number of rounds in MixBits, then the structure of the protocol provides a good avalanche effect; similarly, if we simplify the structure of the protocol but we use MixBits with 32 rounds, then the MixBits function, together with few operations of the protocol, still provides a good avalanche effect.

5 Privacy analysis

We focus our attention on the privacy properties. Several full models, starting from Vaudenay's model [41], are available in the literature. However, since we do not need the full power of them, we consider a simplified model, which enables us to simplify our description as much as possible. Precisely, we define a standard indistinguishability game, essentially as defined by the Juels–Weis model for RFID protocols [22].

5.1 The model

A protocol party P is a $T \in Tags$ or an $R \in Readers$. Tags and readers interact in protocol sessions. Tags are identified by unique IDs. So, to distinguish a tag T_0, with identifier ID_0, from a tag T_1 with identifier ID_1, we need to know the IDs (or at least one of the ID). In other words, to recognize a tag we need to know its identifier. An adversary controls the communication channel between the protocol parties, in a passive or an active mode. The interactions of the adversary with the parties are formally captured by giving it the ability to issue oracle queries. Precisely, an adversary can issue four oracle queries:

- Execute(R, T, i). This query models passive attacks, where the adversary just has read access to an honest execution of the ith session of the protocol between R and T.
- Send(P_1, P_2, i, m). This query models active attacks, allowing the adversary to impersonate party P_1 in the ith session of the protocol and send a message m of its choice to party P_2.
- Corrupt(T, K'). This query allows the adversary to learn the stored secret K of the tag T, and to change it to a new value K' of its choice.
- Test(T_0, T_1, i). This query issues a test query on session i: depending on a randomly chosen bit $b \in \{0; 1\}$, the adversary receives access to T_b from the set $\{T_0, T_1\}$. The adversary succeeds if it can guess correctly the bit b.

The indistinguishability game, $Exp^{ind}_{\mathcal{A}}$, run by a Challenger, is played between an adversary $\mathcal{A}$ and a collection of reader and tag instances. It is composed of three phases:

$Exp_{\mathcal{A}}^{ind}$

1. **Learning:** $\mathcal{A}$ has the ability to send Execute, Send, and Corrupt queries to the Challenger. This phase models the adversary ability to run passive or active attacks, while interacting with reader and tag instances in protocol sessions.
2. **Challenge:**
   (a) At a certain point, $\mathcal{A}$ chooses two fresh tags T_0 and T_1, and sends to the Challenger a Test query with these two tags. Freshness means that to the tags have not been issued Corrupt queries.
   (b) The Challenger randomly chooses a bit $b \in \{0, 1\}$, and $\mathcal{A}$ is given oracle access to T_b, without the knowledge of b and the possibility of a Corrupt query on it. $\mathcal{A}$ cannot also send queries to T_0 and T_1 any more.
   (c) $\mathcal{A}$ continues making any Execute, Send, and Corrupt queries on the other tags.
3. **Guessing:**
   Eventually, $\mathcal{A}$ terminates the game, and outputs a bit b', as its guess of the value of b. If $b' = b$, then the Challenger outputs 1, i.e., $Exp_{\mathcal{A}}^{ind} = 1$. In such a case, we say that the adversary *wins*.Otherwise, $Exp_{\mathcal{A}}^{ind} = 0$.

If $\mathcal{A}$ fails to win the game, it means that it cannot distinguish the tags. Implicitly, it also means that the protocol does not leak private information[3].

A protocol is (t, ϵ)-private if, for any adversary $\mathcal{A}$, which runs for at most t steps, there exists a small ϵ, such that:

$$Pr[Exp_{\mathcal{A}}^{ind} = 1] \leq \frac{1}{2} + \epsilon.$$

5.2 Gossamer indistinguishability game

By using the above privacy model, and exploiting machine learning classifiers, we show that the Gossamer protocol is not (t, ϵ)-private. We exhibit a passive adversary $\mathcal{A}$, that is, an adversary that does not use the Send(P_1, P_2, i, m) and Corrupt(T, K') oracle queries. With such an attack, $\mathcal{A}$ produces labeled tags replies, trains a classifier, and is able to distinguish unlabeled tag replies with a significant accuracy.

Next we describe the details of $\mathcal{A}$, which plays the indistinguishability game $Exp_{\mathcal{A}}^{ind}$.

Algorithm $\mathcal{A}$

1. For t times, $\mathcal{A}$ chooses a pair of target tags, say $T_0^{(z)}$ and $T_1^{(z)}$, with identifiers $ID_0^{(z)}$ and $ID_1^{(z)}$, and issues an Execute$(R, T_0^{(z)}, i)$ query and an Execute$(R, T_1^{(z)}, i)$, for $i = 1, \ldots, n$. This allows him to build pairs with the final reply of the tag (the D values in the protocol) and the identifier $ID_j^{(z)}$. Then, with the above pairs, $\mathcal{A}$ constructs a dataset, and trains a Siamese neural network, which will be used as a classifier to distinguish the tags.
2. $\mathcal{A}$ randomly chooses two fresh tags (T_0, T_1), and handles them to the Challenger. He gets oracle access to T_b.
3. $\mathcal{A}$ sends an Execute$(R, T_b, 1)$ query.Let D_b be the last message sent by T_b. $\mathcal{A}$ uses the Siamese network by giving as input D_0, a reply from one of the Execute(R, T_0, i) queries of the learning phase, and D_b.
4. If the Siamese network outputs a similarity match, then $\mathcal{A}$ outputs 0, otherwise $\mathcal{A}$ outputs 1.

To estimate $\mathcal{A}$'s success probability within the game, the specific dataset that we have used has been constructed by selecting 100 different pairs of tags $T_0^{(z)}, T_1^{(z)}$, $z = 1, \ldots, 100$. Then, for each pair, we ran a set of 1500 Execute$(R, T_0^{(z)}, i)$, and Execute$(R, T_1^{(z)}, i)$, for $i = 1, \ldots, 1500$, obtaining labeled pairs (D, ID), where ID is the identifier of the tag that caused the protocol to create D as the final reply of the tag. Thus, overall, we built a training set with 300,000 labeled objects. These objects represent pairs of executions of the protocol corresponding to pair of tags. The classifier can be trained to distinguish the two tags in each pair.

The specific Siamese network[4] that we have used has been described in Sect. 3.3. Following what has been done in Sects. 4.3.1 and 4.3.2, we have tried with two different approaches for the features vector, using first the direct representation of the object as the actual string of 96 bits and then an alternative representation of the data using as features vector a 95-bit vector as defined in Sect. 4.3.2. Training the Siamese network with the 96-bit feature vector, we obtained an average accuracy[5] of 0.572, while training the network with the 95-bit features vector, we obtained an average accuracy of 0.638. These average values are the result of 10 repetitions of the training and test phases. Table 4 shows the accuracy values obtained in the 10 repetitions.

[3] Such a standard modeling is, of course, highly demanding, and in some applications it might be more than strictly needed. But, as widely agreed within the community, achieving such a notion, enables a safe use of the protocol in any application.

[4] We point out that we tried also with standard classifiers obtaining results supersided by the Siamese network.

[5] Notice that even percentages close to 0.5 could yield some advantages to a distinguisher. However, the bigger the accuracy the stronger is the attack.

Table 4 Gossamer tag distinguishability accuracy with the Siamese network

	1	2	3	4	5	6	7	8	9	10	Avg
FV1	0.57	0.58	0.56	0.57	0.59	0.55	0.55	0.59	0.59	0.57	0.572
FV2	.58	0.6	0.61	0.63	0.66	0.65	0.64	0.67	0.66	0.68	0.638

Features: $FV1 = 96$-bit, $FV2 = 95$-bit

Therefore, by using the best approach for training the Siamese network (0.638 ≈ 0.64), we have that:

$$\begin{aligned} Pr(Exp_{\mathcal{A}}^{ind} = 1) &= Pr(b = 0) \cdot Pr(\mathcal{A}\text{outputs } 0 | b = 0) + \\ &\quad + Pr(b = 1) \cdot Pr(\mathcal{A}\text{outputs } 1 | b = 1) \\ &= \left(\frac{1}{2} + \frac{1}{2}\right) \cdot 0.64 \\ &= 0.50 + 0.14 \\ &> \frac{1}{2} + \frac{1}{10}. \end{aligned}$$

It follows that, with advantage of at least $\frac{1}{10}$ over a random guess, $\mathcal{A}$ succeeds. Hence, Gossamer is not (t, ϵ)-private, for any adversary $\mathcal{A}$ with running time[6] of at least $t = 30{,}000 \approx 2^{15}$ steps.

5.3 Cryptography and neural networks

Our findings of the previous sections have some elements of novelty, which are of independent interest. From a theoretical point of view, the relation between cryptography and machine learning has been investigated since the ’90s, e.g., [37]. Moreover, neural networks have been used as a tool for designing some cryptographic primitives or protocols, e.g., [23,24]; as well as, cryptography and, more precisely, multiparty computation and homomorphic encryption, have been used to set up secure and private implementations of neural networks, e.g., [29,35] (see [7] for a survey). Few applications to cryptanalysis are also known, e.g., [4,26]. Our application is a further example of such a possibility, and of the potentialities that these methods might have in cryptanalysis. Actually, Siamese neural networks, have been extensively used in several fields, as the long list of applications, reported in [12], examplifies. However, no application to cryptography belongs to that list. Our work is the first example. Thus, we feel that such a *similarity-finding* tool deserves an in-depth study: as a future research goal, it would be worthy to test its power in distinguishing other pseudo-random objects, from truly random ones [19].

[6] Notice that, we are assuming implicitly that $\mathcal{A}$ can send an oracle query in each execution step. Perhaps, we could be successful also with a smaller number of oracle queries, but we did not care about possible optimizations, since our goal was only to show that the approach works.

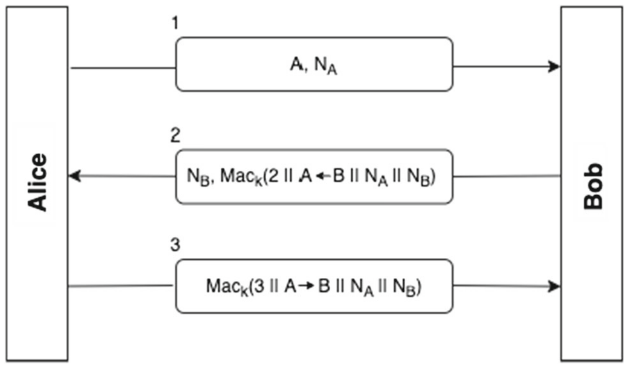
Alice
Bob
1
A, N_A
2
N_B, Mac_K(2 || A←B || N_A || N_B)
3
Mac_K(3 || A→B || N_A || N_B)

Fig. 10 Three-round mutual authentication protocol

6 Performance analysis

To evaluate the performance of the Gossamer protocol, we have done a rough comparison with a standard three-round mutual authentication protocol, based on a symmetric MAC, whose structure is reported in Fig. 10.

We have implemented both Gossamer and four instances of the standard three-round mutual authentication protocol, each of them with a different choice for the MAC computation, and compared the *computational effort* required by each of the implementations, obtained by employing AES, PRESENT, Speck and Simon, respectively. For the first two, AES and PRESENT, we have used public available implementations from github.[7] For AES, we have implemented a small optimization: to save memory usage, we generate the S-box dynamically.

The standard mutual authentication protocol uses the identity of the participant A, directionality bits, i.e., $A \rightarrow B$ and $A \leftarrow B$, counter values, i.e., 2 and 3, and random nonces, N_A and N_B, to deal with standard attacks, e.g., message reordering, message reply, and message reflection. Alice and Bob share the secret key K. We choose the CBC-MAC scheme for computing the MAC, instantiating the four block-ciphers, with the following sizes:

1. *AES*, with a 128-bit block size and key size
2. *Present*, with a 64-bit block size and a 128-bit key size
3. *Speck*, with a 96-bit block size and key size
4. *Simon*, with a 96-bit block size and key size.

[7] https://github.com/kokke/tiny-AES-c, https://github.com/michaelkitson/Present-8bit.

Table 5 Performance results

Scheme [block/key size (b)]	Code size (B)	RAM (B)	Time (cyc.)	Performance
Gossamer	2974	96	381769	18.93
Present (64/128)	1528	56	1513138	43.93
AES (128/128)	1860	68	38800	7.02
Simon (96/96)	1594	64	275384	11.83
Speck (96/96)	1162	48	80592	6.07

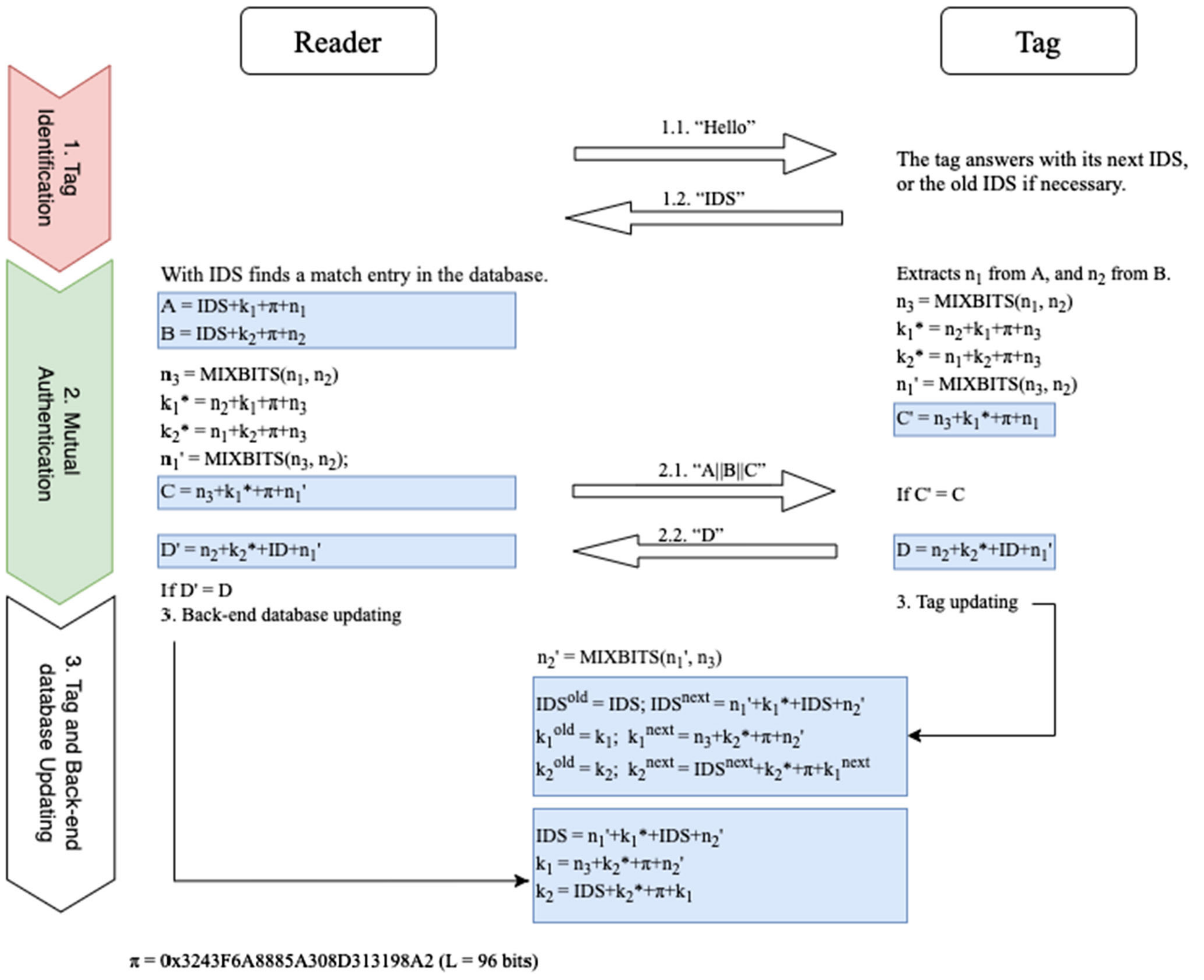

Fig. 11 Gossamer-S1

We associated the tag to Bob in the scheme and evaluated the cost of his computation. Assuming that the counter is represented with 2 bits, the directionality bit with 1 bit, and the nonces with 96 bits, in the second round of the protocol, Bob must generate a CBC-MAC of a 195-bit message, i.e., of $2||0||N_A||N_B$. Then, the message is sent to Alice. Alice replies, and Bob, to authenticate Alice in the third round, has to verify the CBC-MAC of a 195-bit message, i.e., of $3||1||N_A||N_B$. Since the lengths of the messages are not a multiple of the block lengths, the padding (10*) is used to reach the proper lengths.

To test the implementation of the block ciphers and of Gossamer, we used a software platform simulating the Atmel 8-bit AVR microcontroller family. The platform is based on the ATmega128 microprocessor, running at a frequency of 16 MHz, with a flash program memory of 128 Kbytes, and 4 Kbytes of system RAM [14]. More specifically, we used the Atmel Studio 7 integrated development platform [2] to write

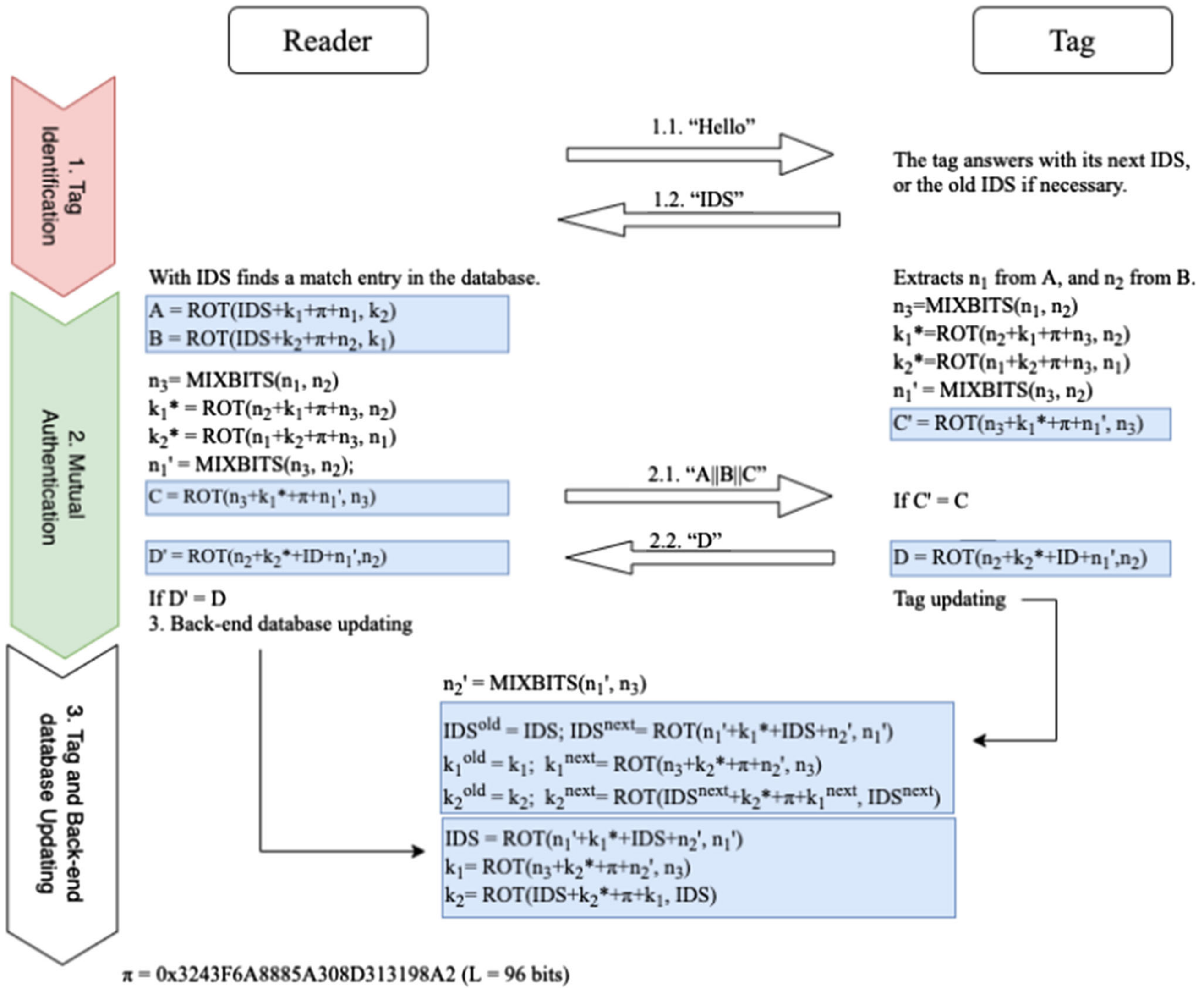
Reader
Tag
1. Tag Identification
2. Mutual Authentication
3. Tag and Back-end database Updating
1.1. "Hello"
1.2. "IDS"
The tag answers with its next IDS, or the old IDS if necessary.
With IDS finds a match entry in the database.
A = ROT(IDS+k1+π+n1, k2)
B = ROT(IDS+k2+π+n2, k1)
n3= MIXBITS(n1, n2)
k1* = ROT(n2+k1+π+n3, n2)
k2* = ROT(n1+k2+π+n3, n1)
n1' = MIXBITS(n3, n2);
C = ROT(n3+k1*+π+n1', n3)
D' = ROT(n2+k2*+ID+n1',n2)
If D' = D
3. Back-end database updating
Extracts n1 from A, and n2 from B.
n3=MIXBITS(n1, n2)
k1*=ROT(n2+k1+π+n3, n2)
k2*=ROT(n1+k2+π+n3, n1)
n1' = MIXBITS(n3, n2)
C' = ROT(n3+k1*+π+n1', n3)
2.1. "A||B||C"
If C' = C
2.2. "D"
D = ROT(n2+k2*+ID+n1',n2)
Tag updating
n2' = MIXBITS(n1', n3)
IDSold = IDS; IDSnext= ROT(n1'+k1*+IDS+n2', n1')
k1old = k1; k1next= ROT(n3+k2*+π+n2', n3)
k2old = k2; k2next= ROT(IDSnext+k2*+π+k1next, IDSnext)
IDS = ROT(n1'+k1*+IDS+n2', n1')
k1= ROT(n3+k2*+π+n2', n3)
k2= ROT(IDS+k2*+π+k1, IDS)
π = 0x3243F6A8885A308D313198A2 (L = 96 bits)

Fig. 12 Gossamer-S2

and run the implementations written in C. The set of metrics, provided by the platform, allows to compute a so-called *merit number*, useful to estimate the performance and rate all the ciphers. More precisely, the three performance metrics take into account the following aspects:

1. *RAM consumption*, measured in byte: the occupied RAM by data section (initialized variables), uninitialized variables, and stack consumption;
2. *code size*, measured in byte: the required flash memory for data section and the cipher code;
3. *execution time*: the number of clock cycles needed to run a cipher.

Denoting, for each metric m of set M, with w_m, v_m, and $min(v_m)$ the *weight*, the *value*, and the *minimum* value among all the ciphers, respectively, the performance estimate is given by the formula:

$$\text{performance} = \sum_{m \in M} w_m \frac{v_m}{\min(v_m)} \tag{4}$$

The execution time weight is set to one, but considering that memory usage is more important for RFID tags, the RAM and code size weights are set to two.

The performance results for the C-language implementation of Gossamer, and of the standard three-round protocol, instantiated with *AES*, *Present*, *Speck*, and *Simon*, are shown in Table 5.

Both Bob's message and the key are kept in RAM in the implementation of the block ciphers. Once the key-schedule operation is done, the round keys are stored in the flash memory, before the MAC is generated. In Gossamer implementation, the values of the keys, k_1, k_2, of the identifiers,

ID, IDS, and of the messages A, B, C, are also be kept in RAM. In order to use the RAM efficiently, whenever it is possible, their values are overwritten during the computations. Besides, the execution time of Gossamer depends on several circular rotations by variable numbers, i.e., k_1, k_2, and the nonces extracted from the messages A and B (mod96). Gossamer is initialized several times with random values for these variables, generated by the `/dev/random/` generator, and the average execution time is measured. The results indicate that, among all the authentication methods, Gossamer has the highest code size as well as the highest RAM usage. If we compare the performance of Gossamer with the three-round protocol where the MAC is implemented through *Speck*, which is the best rated method among the four we have considered, the comparison clearly shows that its performance suffers heavily in code size and execution time. As well as, its large execution time is mostly due to the MixBits function, and the rotation operation. Rama et al. [36] also analyzed the power consumption of Gossamer. They proved that, since the MixBits function and the rotation operation extensively employ the modulo and the shift operations in their computations, these in turn negatively impact on the power consumption.

7 Conclusions

We have briefly surveyed the few known attacks against Gossamer and, then, analyzed the structure of the protocol. We have shown that the *MixBits* function does not realize a pseudorandom function, even in a weak form. Then, we have employed artificial intelligence techniques, in order to defeat the privacy features of the protocol, and we have shown that tags can be distinguished with non-negligible probability. We feel that the results obtained with the AI techniques encourage to further study the applicability of such techniques to certain cryptographic protocol analysis. Finally, we have compared the performance of Gossamer with a standard three-round mutual authentication protocol, based on MACs, implemented through *AES, Present, Simon* and *Speck*, and we have shown that the computational effort required by Gossamer does not provide a saving, compared to the lightweight implementations of the standard three-round mutual authentication protocol. Our software implementation and evaluation, with all the limits that brings with it, e.g., an optimized hardware implementation of Gossamer could gain in terms of circuitry costs and efficiency, suggests that standard cryptographic protocols, instantiated with lightweight and sufficiently well-scrutinized primitives, perhaps could be a suitable better option to deal with the new challenges, posed by the heavily constrained devices, which populate our current computational environment.

Acknowledgements We would like to thank Dr. Laura Zollo for helping us, in a preliminary version of this work, during her final-degree project for the Master Program in Computer Science (Laurea Magistrale in Informatica). Moreover, we are grateful to two anonymous referees for comments and suggestions, which helped us to significantly improve the quality of the paper.

Funding Not applicable. This research has been carried out by the authors as part of their normal duties as professors/students.

Availability of data and materials Not applicable.

Declarations

Conflict of interest/Competing interests. The authors declare that there are no conflict of interest nor competing interests.

Research involving human participants and/or animals This article does not contain any studies with human participants or animals performed by any of the authors.

Informed consent Not applicable.

Code availability Not applicable.

References

1. https://docs.rapidminer.com/
2. https://www.microchip.com/mplab/avr-support/atmel-studio-7
3. Ahmed, E.G., Shaaban, E., Hashem, M.: Lightweight mutual authentication protocol for low cost RFID tags. arXiv:1005.4499 (2010)
4. Alani, M.M.: Neuro-cryptanalysis of des and triple-des. In: Huang, T., Zeng, Z., Li, C., Leung, C.S. (eds.) Neural Information Processing, pp. 637–646. Springer, Berlin (2012)
5. Avoine, G., Carpent, X., Hernandez-Castro, J.: Pitfalls in ultralightweight authentication protocol designs. IEEE Trans. Mob. Comput. **15**(9), 2317–2332 (2015)
6. Avoine, G., Carpent, X., Martin, B.: Strong authentication and strong integrity (sasi) is not that strong. In: International Workshop on Radio Frequency Identification: Security and Privacy Issues, pp. 50–64. Springer (2010)
7. Azraoui, M., Bahram, M., Bozdemir, B., Canard, S., Ciceri, E., Ermis, O., Masalha, R., Mosconi, M., Önen, M., Paindavoine, M., Rozenberg, B., Vialla, B., Vicini, S.: SoK: Cryptography for Neural Networks, pp. 63–81. Springer, Cham (2020)
8. Bilal, Z., Martin, K.: Ultra-lightweight mutual authentication protocols: weaknesses and countermeasures. In: 2013 International Conference on Availability, Reliability and Security, pp. 304–309. IEEE (2013)
9. Bilal, Z., Masood, A., Kausar, F.: Security analysis of ultra-lightweight cryptographic protocol for low-cost rfid tags: Gossamer protocol. In: 2009 International Conference on Network-Based Information Systems, pp. 260–267. IEEE (2009)
10. Bromley, J., Guyon, I., LeCun, Y., Säckinger, E., Shah, R.: Signature verification using a siamese time delay neural network. Adv. Neural Inf. Process. Syst. **6**, 737–744 (1993)
11. Carpent, X., DArco, P., De Prisco, R.: Ultralightweight authentication protocols. Selected Topics in Security of Ubiquitous Computing Systems (2019). ISBN: 978-3-030-10591-4

12. Chicco, D.: Siamese neural networks: an overview. Artif. Neural Netw., pp. 73–94 (2021)
13. Chien, H.Y.: SASI: a new ultralightweight RFID authentication protocol providing strong authentication and strong integrity. IEEE Trans. Dependable Secure Comput. **4**(4), 337–340 (2007)
14. Cooperation, A.: 8-bit AVR microcontroller with 128k bytes in-system programmable flash (2007)
15. DArco, P.: Ultralightweight cryptography. In: International Conference on Security for Information Technology and Communications, pp. 1–16. Springer (2018)
16. DArco, P., De Prisco, R.: Design weaknesses in recent ultralightweight RFID authentication protocols. In: IFIP International Conference on ICT Systems Security and Privacy Protection, pp. 3–17. Springer (2018)
17. DArco, P., De Santis, A.: Weaknesses in a recent ultra-lightweight RFID authentication protocol. In: International Conference on Cryptology in Africa, pp. 27–39. Springer (2008)
18. DArco, P., De Santis, A.: On ultralightweight RFID authentication protocols. IEEE Trans. Dependable Secure Comput. **8**(4), 548–563 (2010)
19. Fan, F., Wang, G.: Learning from pseudo-randomness with an artificial neural network? Does god play pseudo-dice? IEEE Access **6**, 22987–22992 (2018). https://doi.org/10.1109/ACCESS.2018.2826448
20. Hernandez-Castro, J.C., Estevez-Tapiador, J.M., Ribagorda-Garnacho, A., Ramos-Alvarez, B.: Wheedham: an automatically designed block cipher by means of genetic programming. In: 2006 IEEE International Conference on Evolutionary Computation, pp. 192–199. IEEE (2006)
21. Hernandez-Castro, J.C., Tapiador, J.M., Peris-Lopez, P., Quisquater, J.J.: Cryptanalysis of the SASI ultralightweight RFID authentication protocol with modular rotations. arXiv:0811.4257 (2008)
22. Juels, A., Weis, S.A.: Authenticating pervasive devices with human protocols. In: Annual International Cryptology Conference, pp. 293–308. Springer (2005)
23. Kanter, I., Kinzel, W., Kanter, E.: Secure exchange of information by synchronization of neural networks. Europhys. Lett. (EPL) **57**(1), 141–147 (2002)
24. Karras, D., Zorkadis, V.: Strong pseudorandom bit sequence generators using neural network techniques and their evaluation for secure communications. In: McKay, B., Slaney, J. (eds.) AI 2002: Advances in Artificial Intelligence, pp. 615–626. Springer, Berlin (2002)
25. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. CRC Press, Boca Raton (2021)
26. Klimov, A., Mityagin, A., Shamir, A.: Analysis of neural cryptography. In: Zheng, Y. (ed.) Advances in Cryptology—ASIACRYPT 2002, pp. 288–298. Springer, Berlin (2002)
27. Koch, G., Zemel, R., Salakhutdinov, R.: Siamese neural networks for one-shot image recognition. In: ICML Deep Learning Workshop, vol. 2. Lille (2015)
28. Kotu, V., Deshpande, B.: Predictive Analytics and Data Mining: Concepts and Practice with Rapidminer. Morgan Kaufmann, Burlington (2014)
29. Orlandi, C., Piva, A., Barni, M.: Oblivious neural network computing via homomorphic encryption. EURASIP J. Inf. Secur. (037343) (2007)
30. Peris-Lopez, P., Hernandez-Castro, J.C., Estevez-Tapiador, J.M., Ribagorda, A.: Emap: an efficient mutual-authentication protocol for low-cost RFID tags. In: OTM Confederated International Conferences “On the Move to Meaningful Internet Systems”, pp. 352–361. Springer (2006)
31. Peris-Lopez, P., Hernandez-Castro, J.C., Estévez-Tapiador, J.M., Ribagorda, A.: Lmap: a real lightweight mutual authentication protocol for low-cost RFID tags. In: Proc. of 2nd Workshop on RFID Security, vol. 6 (2006)
32. Peris-Lopez, P., Hernandez-Castro, J.C., Estevez-Tapiador, J.M., Ribagorda, A.: M 2 ap: a minimalist mutual-authentication protocol for low-cost RFID tags. In: International Conference on Ubiquitous Intelligence and Computing, pp. 912–923. Springer (2006)
33. Peris-Lopez, P., Hernandez-Castro, J.C., Tapiador, J.M., Ribagorda, A.: Advances in ultralightweight cryptography for low-cost RFID tags: Gossamer protocol. In: International Workshop on Information Security Applications, pp. 56–68. Springer (2008)
34. Phan, R.C.W.: Cryptanalysis of a new ultralightweight RFID authentication protocol-SASI. IEEE Trans. Dependable Secure Comput. **6**(4), 316–320 (2008)
35. Pinkas, B.: Cryptographic techniques for privacy-preserving data mining. ACM Sigkdd Explor. Newsl. **4**(2), 12–19 (2002). https://doi.org/10.1145/772862.772865
36. Rama, N., Suganya, R.: Ssl-map: a more secure gossamer-based mutual authentication protocol for passive RFID tags. Int. J. Comput. Sci. Eng. **2**, 363–367 (2010)
37. Rivest, R.L.: Cryptography and machine learning. In: H. Imai, R.L. Rivest, T. Matsumoto (eds.) Advances in Cryptography—ASIACRYPT ’91, Lecture Notes in Computer Science, vol. 739, pp. 427–439. Springer
38. Sun, H.M., Ting, W.C., Wang, K.H.: On the security of Chiens ultralightweight RFID authentication protocol. IEEE Trans. Dependable Secure Comput. **8**(2), 315–317 (2009)
39. Tagra, D., Rahman, M., Sampalli, S.: Technique for preventing dos attacks on RFID systems. In: SoftCOM 2010, 18th International Conference on Software, Telecommunications and Computer Networks, pp. 6–10. IEEE (2010)
40. Van Deursen, T., Radomirovic, S.: Attacks on RFID protocols. IACR Cryptol. ePrint Arch. **2008**(310), 1–56 (2008)
41. Vaudenay, S.: On privacy models for RFID. In: International Conference on the Theory and Application of Cryptology and Information Security, pp. 68–87. Springer (2007)
42. Yeh, K.H., Lo, N.: Improvement of two lightweight RFID authentication protocols. Inf. Assur. Secur. Lett. **1**(1), 6–11 (2010)