Annales UMCS Informatica AI XII, 4 (2012) 11–22
DOI: 10.2478/v10065-012-0018-y

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI
http://www.annales.umcs.lublin.pl/

Hardened Bloom Filters, with an Application to Unobservability

Nicolas Bernard[1*], Franck Leprévost[1†]

[1] *LACS, University of Luxembourg*
162 A, Avenue de la Faïencerie, L-1511 Luxembourg

Abstract – Classical Bloom filters may be used to elegantly check if an element e belongs to a set S, and, if not, to add e to S. They do not store any data and only provide boolean answers regarding the membership of a given element in the set, with some probability of false positive answers. Bloom filters are often used in caching system to check that some requested data actually exist before doing a costly lookup to retrieve them. However, security issues may arise for some other applications where an active attacker is able to inject data crafted to degrade the filters' algorithmic properties, resulting for instance in a Denial of Service (DoS) situation. This leads us to the concept of *hardened Bloom filters*, combining classical Bloom filters with cryptographic hash functions and secret nonces. We show how this approach is successfully used in the TrueNyms unobservability system and protects it against replay attacks.

1 Introduction

Many applications in computer science depend on the result of the following problem: check if an element e belongs to a set S, and, if it does not, add e to S. Depending on the application we have in mind, the "match" or "no match" answer will usually lead to additional processing, like for instance in the following two examples:

(1) Filtering duplicated packets on a network connection: On a network connection, it can happen that a packet is duplicated. The destination host then receives it twice, so does the application. This is for instance the case on a UDP connection.

*Nicolas.Bernard@uni.lu
†Franck.Leprevost@uni.lu

(2) Counting the number of different elements in a collection: If they are not in this set, a counter is increased and the element is added to the set.

Bloom filters [**1**] address these problems in an elegant manner. A Bloom filter is a probabilistic data structure that allows to represent a finite set S without storing the actual elements of the set S. Among their main properties, Bloom filters have small footprints, a fast lookup time, allow to add elements quickly to the represented set S, and the addition of an element cannot fail due to the data structure being "full". Bloom filters do not store any data and can only provide boolean answers on the membership of a given element in the set, with some probability of false positive answers. They are often used in caching system to check that some requested data actually exist before doing a costly lookup to retrieve them.

In the situation of the example (1) above, a Bloom filter at the receiving end could be used to drop the duplicated packets: packets that do not match are processed (*i.e.*, used by the application) and added to the set, while packets that do match are considered as duplicated and discarded.
In the situation of example (2), each element of the collection is matched against a Bloom filter representing an "already accounted" set. While the result is only of probabilistic nature, its complexity is $O(m)$ whereas the complexity of a classical algorithm remains $O(m \log m)$, where m is the number of elements of the collection.

This being said, security issues may be raised for many applications, leading *e.g.* to Denial of Service (DoS) attacks. The purpose of this article is to provide a solution to these issues by introducing *hardened Bloom filters*. Moreover, we show their use in the seminal example of the TrueNyms protocol [**2**], which raised our interest in Bloom filters and motivated the present contribution.

This article is organized as follows: in section 2, we briefly explain the underlying concept of a classical Bloom filter. In section 3, we describe the security issues that an external malicious party may exploit, leading to the construction of hardened Bloom filters. In section 4, we briefly describe the TrueNyms unobservability system, and describe how to efficiently use hardened Bloom filters to prevent replay attacks on this system. We conclude this article with some further ideas for the enhancements of our approach, which we plan to develop in due time.

2 Classical Bloom filters

A Bloom filter (in the classical understanding as defined in [**1**]) is a probabilistic data structure representing a finite set S. It consists of a bit array A of size 2^n (in practice n is small, say $n < 25$), and k distinct hash functions $(H_j)_{1 \leq j \leq k}$ such that

$$H_j(data) = i_j \in [0, 2^n - 1]. \tag{1}$$

In other words, i_j is an index of A, depending on the *data* considered. Moreover, k is also small: its chosen value — in a first approach — depends on the allowed probabilistic "false-positive" occurrences according to formula 2 below. The discussion about the (lack of) requirements on hash functions in the context of classical Bloom filters is addressed in part 2.2.

2.1 Construction of S and A

Initially, $S = \emptyset$ and all the bit values of A are equal to 0. An element e is added to S by setting to 1 all the positions of the array A indexed by the hash values $i_1 = H_1(e), i_2 = H_2(e), \ldots, i_k = H_k(e)$:

$$\forall j \in [1, k],\ A[H_j(e)] \leftarrow 1.$$

The test to determine if an element e is already in S is performed by generating the indices for this element. An element e is then probably in S if, and only if:

$$\forall j \in [1, k],\ A\left[H_j(e)\right] = 1.$$

The probability in the previous sentence applies only to the "if" part. Indeed, there can be values i, j, e, e' s.t.

$$H_i(e) = H_j(e').$$

In other words, an index in the array A could be "part of" multiple elements of S. As a consequence, there is no way to remove elements from S and, once set to 1, a value $A[i]$ is never reset to 0. It implies in particular that, once added, an element belonging to S is always found if matched against the filter.

Now, with some probability, the filter can represent an element e as belonging to S although it is not the case: it may indeed happen that all the indices corresponding to e are equal to 1, while $e \notin S$. Such a "false-positive" occurs with a probability:

$$\left(1 - \left(1 - \frac{1}{2^n}\right)^{km}\right)^k \approx \left(1 - e^{\frac{-km}{2^n}}\right)^k, \tag{2}$$

where m is the number of elements in S.

2.2 Non-cryptographic hash functions

A hash function as used in the context of classical Bloom filters *a priori* differs strongly from a hash function used in the context of cryptology. It is a function[1]:

$$H : \mathbb{N} \longrightarrow [0, 2^n - 1]$$

with good statistical distribution properties for given "normal data", as described for instance in section 6.4 of [**3**]. In particular, these hash functions usually lack the

[1]We consider here any finite word on any finite alphabet as mappable to an element of $\mathbb{N}$, and that distinct words lead to distinct elements of $\mathbb{N}$.

compression property (see [**4**, section 9.2.1]) that is a mandatory and important part of a cryptographic hash function.

Such a hash function can be very simple, and usually it is in order to be fast. For instance, it may consist in the modular division by some prime, chosen according to the needed size of the image. In fact, since the hash function does not need to consider all the data given but only a suitable part to obtain a correct distribution, we can even construct hash functions with complexity in $O(1)$.

Consequences are multiple, but here we will only note the three following :

(1) Recall that we need k distinct hash functions for the classical Bloom filters. We can create many different functions with similar properties by changing a parameter in one fixed scheme. For instance, in a scheme based on modular division, the choice of k distinct appropriate primes leads to k distinct hash functions.

(2) It is possible to find preimages : it means that given an i, it is possible to find $D_x, D_y, \cdots$ such that $H(D_x) = H(D_y) = \cdots = i$. Indeed, many simple hash functions can be easily inverted. Anyway, given the usual size of the image set, it would be easy to find such values by brute-force.

(3) It is usually even possible, given a few such hash functions $H_1, \cdots, H_j$ and corresponding indices $i_1, \cdots, i_j$, to find a common preimage D such that

$$H_1(D) = i_1, \cdots, H_j(D) = i_j. \tag{3}$$

3 Security issues and Hardened Bloom filters

As mentioned in the introduction (section 1), security issues may be raised in some applications. For instance, assume the elements to be matched can be tampered by an external malicious party, say Mallory. Recall then that the probability given in equation 2 applies to "ordinary" elements. Since the hash functions H_k are *a priori* non-cryptographic ones, Mallory can craft *special elements* that will fill A with bits set to 1 much faster than random data would[2]. Of course, once all the bits of the array A are equal to 1, each element tried against the filter will match, which results in a denial of service (DoS) attack in the cases given beforehand: all the elements are considered as already in the set, even when they are not. So, Bloom filters must be hardened to prevent such attacks if Mallory controls the incoming data.

If an attacker can inject as many elements he wants to, the battle is lost because even if he is restricted to the probability given by equation 2, with m growing, the probability will converge to 1. However, such a case is rare, and most of the time the attacker will find himself unable to add more than a fixed number of elements per time unit. Here, it is possible to fight back, and design appropriate countermeasures.

[2]The irony being that, while collisions are usually a sign of weakness in cryptographic hash functions, here Mallory has to find non-colliding elements in order to set to 1 all the bits of the array A as fast as possible.

3.1 Protection against index selection attacks

To prevent Mallory from just deciding upon a set of indices and creating suitable data to send, the first idea is to use Bloom filters where the k hash functions have some cryptographic properties.

Notably, it must be hard — no faster way than brute force — to find preimages, to insure that the attacker will not be able in practice to find a common preimage as defined in equation 3. With such hash functions, it would be far harder for Mallory to find non-colliding packets than simply deciding which bits in the array A he wants to set and generating the corresponding data.

The natural choice for a hash function with such cryptographic properties, is to take a cryptographic hash function H^c [**4**, page 323]. Note however that the properties of a cryptographic hash function are a superset of what is actually needed: we comment on these aspects in section 5.

3.2 k cryptographic hash functions ?

The first difficulty is to find k such functions. As we have seen in section 2.2, it is easy to have many non-cryptographic hash functions. Unfortunately, even for a small relevant k, we cannot find k different standard cryptographic hash functions. The list of such hash functions mainly consist of MD5, SHA-1, the SHA-2 and RIPEMD families [**4, 5**], and this list can hardly be extended much further.

Nonetheless, there are multiple ways to solve this issue :

(1) Conceptually, the easiest way is probably to add the index of the function before the data. In other words, given one cryptographic hash function H^C, and using the | symbol for concatenation, we define the k hash functions as

$$H_i(data) := H^c\left(i|data\right), \quad 1 \leq i \leq k.$$

Some variants of this method can be imagined. For instance, the index could be used in the initialization vector of the compression function of the hash function. However this proposal only makes the implementation harder as specifying this vector is usually not possible through the API of the cryptographic libraries providing such functions.

(2) One can also think of using the iterated application of the cryptographic hash function H^c to produce the $(H_i)_{1\leq i\leq k}$. More precisely, the k hash functions are defined as

$$H_i(data) := \left(H^c\right)^i(data), \quad 1 \leq i \leq k,$$

with

$$\left(H^c\right)^i(data) = \begin{cases} H^c(data) & \text{if } i = 1, \\ H^c\left(\left(H^c\right)^{i-1}(data)\right) & \text{if } 2 \leq i \leq k. \end{cases}$$

(3) Another way, is to notice that the fingerprint returned by a cryptographic hash function is a lot longer than an index for the bit array of the Bloom

filter. Indeed, the shortest fingerprints are at least 128 bits long, while it is unusual for an index to be more than 25 bits long, as noted in section 2. The idea then would be to see the fingerprint provided by a cryptographic hash function as the concatenation of l indices :

$$H^c(data) = i_1|i_2|i_3|\cdots|i_l|r,$$

where r is an unused "remainder" if the size of the fingerprint is not a multiple of the size of an index, and i_j are the indices of equation 1. Of course, it may happen that $l < k$, then this scheme would need to be combined with one of the previous two to generate the k required indices. However, as there are standard hash functions with fingerprints size up to 512 bits at least, it should be possible to use it alone in most cases.

(4) Another possibility that we will not detail here would be to construct custom hash functions using block ciphers [**4**, section 9.4.1].

Security-wise, there is no evidence that one of the previous schemes has some obvious advantage over the others. Let us then compare them on their speed. The algorithmic complexity of a cryptographic hash function is at least in $O(s)$, where s is the size of the data to be hashed. To simplify, assume that the algorithm complexity of the cryptographic hash function is indeed s, the complexity of the different schemes would then be in :

(1) ks for the first one, as the H^c function is called k times on data of size $s + \epsilon$ (ϵ being the size of the index added before the actual data).

(2) $s + (k-1)f$ for the second one, where f is the size of a fingerprint: H^c is called once on data of size s, then $k-1$ times on the fingerprint of size f generated at the previous step. The second scheme is hence faster than the first one if the data size is large.

(3) The third one needs only one call to the cryptographic hash function if $l \geq k$. If $l < k$ the exact complexity depends on the combination with one of the other schemes, but will be reduced compared to it anyway.

The third scheme then seems to be the best choice, since it is the fastest one. It must be noted however that a cryptographic hash function is anyway much slower than a non-cryptographic one. To take an example, the number of operations to hash data of size s can be as low as 1 for a non-cryptographic hash function as described in 2.2, while it would be of the order of $160s$ for a typical cryptographic hash function like RIPEMD-160 [**6**].

3.3 Protection against offline attacks

Let us recall that the hash functions considered here give a value that is an index for the array A, *i.e.* a value belonging to $[0, 2^n]$, with $n < 25$, and hence preimages can be found by brute force. Moreover, because Bloom filters are deterministic (and the different schemes presented in 3.2 do not change this), the same input will fill two filters in the same way. Mallory can then perform the following offline DoS attack:

Brute force the hash functions to create a set of elements that would fill the Bloom filter faster than "normal" data would. Even if he is not anymore able to select indices and craft data to set them specifically, he can still generate a lot of data packets and send the group of them that sets the greatest number of bits in the array A. While such elements would have some collisions on indices, they would still fill the filter a lot faster than the statistical probability predicts.

Let us summarize the situation: to insure the protection against index selection attacks (seen in part 3.1), we rely on Bloom filters using cryptographic hash functions. Now, to furthermore insure the protection against an offline attack as described above, we add the utilization of secret nonces. A nonce is a random value, which in our context is generated at the instantiation of a Bloom filter and is then used as a key so that the cryptographic hash functions are in fact replaced by MACs (or keyed hash functions, see [**4**, page 325]). Instead of giving all details, we provide here the conceptual idea, which amounts to specializing H^c for each Bloom filter $\mathfrak{F}$ in something like

$$H^{c,\mathfrak{F}}(data) := H^c(n_{\mathfrak{F}}|data), \tag{4}$$

where $n_{\mathfrak{F}}$ is the nonce used for filter $\mathfrak{F}$.

With such a scheme, Mallory is blinded: he is not able to know the effect of an element and hence cannot craft *special elements* anymore. As a consequence, an active DoS attack by Mallory against Bloom filters hardened this way does not work, provided that Mallory is only able to add a limited number of elements per second.

The main drawback is that it is not possible anymore to take the union of two sets by using a bit-wise OR operation on the arrays of the corresponding bloom filters unless they are using the same nonce. For most applications, this however should not be a significant issue.

We define here a *hardened Bloom filters* as a classical Bloom filter using cryptographically-enhanced hash functions together with a secret nonce, addressing index-selection attacks as well as offline attacks.

4 Hardened Bloom filters and TrueNyms

We now describe how such hardened Bloom filters are used in the TrueNyms unobservability system [**7, 8, 2**] as a protection against some forms of active traffic analysis. Let us first recall what TrueNyms[3] is.

[3]We partially rely on [**8**] for the wording of some paragraphs of subsections 4.1 and 4.2, as well as for the figures 1 and 2.

4.1 The TrueNyms unobservability system

The TrueNyms system allows Alice and Bob to communicate over an IP network without any observer knowing it. More precisely, when parties are using TrueNyms for their communications, an observer, as powerful as he may be, is unable to know who they are communicating with. He is unable to know when a communication occurs. He is even unable to know if a communication occurs at all.

This TrueNyms system is a peer-to-peer overlay network based on Onion-Routing [**9, 10**], to which it adds protection against all forms of traffic analysis, including replay attacks. Its performance is experimentally validated and is appropriate for most uses (*e.g.* Web browsing and other HTTP-based protocols like RSS, Instant Messaging, file transfers, audio and video streaming, remote shell, ...) but the usability of applications requiring a very low end-to-end latency (like for instance telephony over IP) may be degraded.

Briefly, Onion-Routing transmits data through nested encrypted tunnels established through multiple relays R_1, R_2, etc. (see Figure 1 — in the following, a *node* denotes either a relay or Alice or Bob). These relays accept to take part in an anonymity system, but are not supposed trusted. Indeed, some of them can cooperate with a passive observer Eve or with an active observer Mallory. Relays see only enciphered traffic and know only the previous and next nodes on the route. They do not know if those nodes are other relays or end-points.

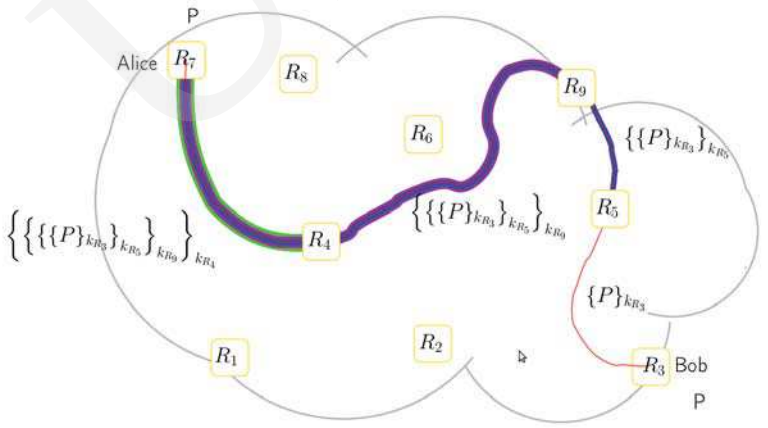

Fig. 1. In Onion-Routing, to communicate with Bob, Alice creates a set of nested encrypted tunnels. For every packet, each relay removes the outermost encryption layer (hence the name of this scheme).

To clarify some terminology used throughout this section, an encrypted tunnel between Alice and one of the nodes is called a *connection*. Then, a set of nested connections between Alice and Bob is called a *route*. Despite being created by Alice, those routes are

not related to IP source routing or other IP-level routing. Standard IP routing is still used between successive nodes if these nodes are on an IP network as we consider here. At last, in TrueNyms, a *communication* is a superset of one or more routes between Alice and Bob that are used to transmit data between them. A communication can use multiple routes simultaneously and / or sequentially.

4.2 Replay attacks

An issue with standard cryptography modes when used in Onion-Routing is that they allow an active replay attack[4]. Let us examine the situation at a relay at a given time: for instance, let us assume that this specific relay is a part of three routes, as depicted in Figure 2.

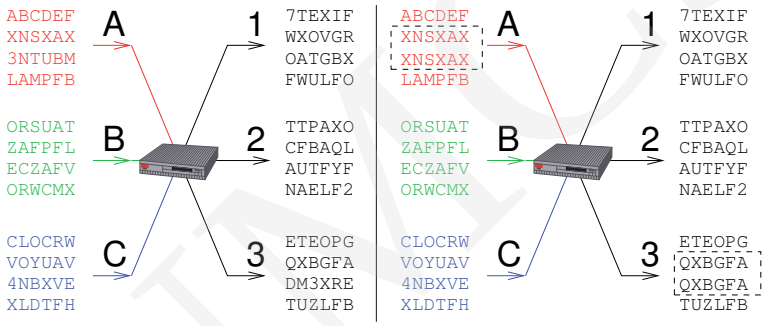

Fig. 2. Cryptography hides connection bindings to a passive observer (left), but not to an active observer able to inject duplicate packets (right).

On the left of Figure 2, the observer sees three distinct incoming connections (A, B, C), while there is also three outgoing connections (1, 2, 3). To make the relaying useless, the observer must discover the relationship between the incoming and the outgoing connections, or at least he must discover the outgoing connection corresponding to an incoming one he is interested in.

As an encryption layer is removed on each connection, he cannot discover this by a casual glance at the content of the packets. Moreover, in TrueNyms, the packet size and rate are normalized, and care is taken to prevent information leaks when a route is established or closed (as described in [**7, 8, 2**]).

Those standard traffic analysis methods are hence closed to an attacker.

However, as cryptography is deterministic, if nothing is done, a given packet entered twice through a same incoming connection would be output twice — in its form with an encryption layer removed — on the corresponding outgoing connection. So Mallory takes a packet and duplicates it, say on connection A, which leads to the right side of Figure 2. He then looks for two identical packets on the output, and finds them

[4]This is different of the replay attacks well known in cryptography, where an attacker can play part of a protocol back from a recording, and that are usually prevented by the use of nonces or timestamps.

on the connection 3, so he learns that connection A and connection 3 are part of the same route. Obviously, depending on the interest of Mallory, he can perform a similar attack on the next relay having the connection 3 as an incoming connection, and then see where it leads ultimately. Or he can perform the same attack on the other incoming connections B and C, and figure out exactly which outgoing connection 1 or 2 corresponds to them.

The obvious way to prevent an external attacker to inject packets would be to use node-to-node authentication on a route, but in this case it would not be sufficient since, even if we assume that the replay of an authenticated packet is not possible, the possibility for Mallory to operate a node must also be accounted for. This means there is no way to actually prevent packet injection by an active observer, and so the system has to be designed in a way that makes such injection useless.

4.3 Using hardened Bloom filters to prevent replay attacks

Recall that packets between two successive nodes on a route can be replayed by Mallory, and hence will be output on the corresponding outgoing connection to the downstream relay.

In the TrueNyms implementation, to prevent such replay attacks, a relay "remembers" all the packets of a transmission and compares each incoming packet on the same connection to them. If it does not match, the packet is forwarded; if it does match, it is dropped (and a dummy packet is forwarded).

Of course this approach requires a very fast way to compare a new packet to the previous ones, hence the need for Bloom filters.

The situation is then similar to the context described in the example (1) of section 1: an accepted packet is added to the filter if it "was not" already in it. In TrueNyms, as the traffic is shaped, Mallory cannot simply flood the filter as the addition to the filter is only done for *transmitted* packets, and packets outside the shaping envelope are simply dropped.

In order to protect our unobservability system against the security issues raised in section 3, TrueNyms relies on hardened Bloom filters.

Notice that, as false positives can occur, legitimate packets may be dropped. This may slightly alter the performance of the system, but is not otherwise an issue as TrueNyms provides end-to-end reliability if needed: the packet will then be resent with another aspect. To ensure this different aspect, unacknowledged packets are buffered unencrypted. If it is necessary to retransmit a packet, a nonce (unrelated to the nonces used in the hardened Bloom filters in part 3.3) it includes is changed before the packet is re-encrypted. As the cipher is used in bi-IGE mode (see below), the new encrypted packet will have no similarities with the old one.

Nonetheless, a long term connection would start to swamp the hardened Bloom Filter after some time, and packets would start to be lost more and more. In TrueNyms, this is not an issue due to two distinct features :

(1) Even if the *communication* is long-term, this is not the case of the *routes* it uses. The lifetime of a route is chosen at random and is fixed before it is used ;
(2) Routes are re-keyed from time to time. It means the encryption keys used for the connections are changed. As the same packet entering twice but going through the encryption layer with different keys would give different (and *a priori* unmatchable without knowing the keys) outputs, the hardened Bloom filters can be replaced by new ones during the key changes.

Of course, it only prevents Mallory from replaying *identical* packets. If let unhindered, he will replay slightly different packets and his attack would be successful because after adding or removing an encryption layer with a standard block cipher mode, the original and replayed packets will have similarities. For the use of hardened Bloom filters to be effective, this attack must be prevented too, for instance by employing a special mode like bi-IGE (which is a bi-directional application of the Infinite Garble Extension mode — Campbell, 1977, [**11**]) as it is done in TrueNyms.

5 Conclusions and further work

In this paper, after recalling the functioning and the main properties of classical Bloom filters, we considered the situation where a malicious party may develop index-selection attacks or offline attacks against some applications, leading *e.g.* to Denial of Service situations. We then designed *hardened Bloom filters* able to withstand such attacks, combining classical Bloom filters together with cryptographic hash functions and secret nonces. Although these hardened Bloom filters are slower than classical Bloom filters, mostly due to the use of cryptographic hash functions over non-cryptographic ones, we described how they are concretely successfully used in the TrueNyms unobservability system to defend it against active traffic analysis attacks.

Should the need arise, performance can probably be improved by further work on the hash functions. Our proposed *hardened Bloom filters* relies notably on cryptographic hash functions. However, the requirements are probably weaker: for instance, while compression and preimage resistance appear to be needed, it is not obvious that second-preimage and collision-resistance are necessary as well. It may hence be possible to construct custom hash functions with only the mandatory properties, that would be faster than the usual cryptographic hash functions. We intend to study these possibilities in a future work.

Finally, multiple variants of Bloom filters have been proposed (*Bloomier filters*, etc.) over the years, some faster, some using less space, some allowing to remove elements,

etc. In a future work, we also intend to study the possibility to similarly harden some of these numerous existing variants of Bloom filters.

Acknowledgements

The FNR/04/01/05/TeSeGrAd grant partially supported this research.

References

[1] Bloom B. H., Space/time trade-offs in hash coding with allowable errors, Communications of the ACM 13 (7) (1970): 422.
[2] Bernard N., Leprévost F., Unobservability of low-latency communications: the TrueNyms protocol, work in progress.
[3] Knuth D. E., Sorting and Searching,The Art of Computer Programming 3 (1998).
[4] Menezes A. J., van Oorschot P. C., Vanstone S. A., Handbook of Applied Cryptography, Discrete Mathematics and its Applications, CRC Press (1997).
[5] Anderson R., Security Engineering: A Guide to Building Dependable Distributed Systems, Wiley (2001).
[6] Preneel B., Dobbertin H., Bosselaers A., The Cryptographic Hash Function RIPEMD-160, CryptoBytes 3 (2) (1997): 9.
[7] Bernard N., Non-observabilité des communications à faible latence, Université du Luxembourg, Université de Grenoble 1 – Joseph Fourier (2008).
[8] Bernard N., Leprévost F., Beyond TOR: The TrueNyms Protocol, Security and Intelligent Information Systems 7053 (2012): 68.
[9] Goldschlag D. M., Reed M. G., Syverson P. F., Hiding Routing Information, Proceedings of Information Hiding: First International Workshop, Springer-Verlag, LNCS 1174 (1996): 137.
[10] Reed M. G., Syverson P. F., Goldschlag D. M., Anonymous connections and Onion Routing, IEEE Journal on Selected Areas in Communications 16(4) (1998): 482.
[11] Knudsen L., Block Chaining Modes of Operation, Department of Informatics, University of Bergen (2000); http://www.ii.uib.no/publikasjoner/texrap/ps/2000-207.ps