FINAL INTERNSHIP

Robust Privacy-Preserving Gossip Averaging

Amaury Bouchra Pilet

École Normale Supérieure « Ulm »
INRIA Rennes - Bretagne Atlantique

Ventôse - Messidor 226

Contents

I Introduction

With the recent evolution in the applications of information technology like the Internet of Things, there is a strong need for protocols enabling a large network of devices to perform computation on data originating from every device. For these protocols, privacy protection is an important property to have, since the data processed may be personal.
This kind of distributed data processing is commonly done with a class of algorithm called Gossip Algorithms.

I.1 Gossip algorithm and privacy

Gossip algorithms remove the needs of a central server, also removing the risk of being spied by the server's operator or him being forced to give personal information to some kind of "security" agency. While this is a good thing for privacy protection, these algorithms also have serious disadvantages for privacy. They require users to send information to unknown peers, which may allow, not only big companies or governmental agencies, but also criminal organizations or "curious" people to access personal data rather easily, compared to having to violate heavily secured servers and communications. The purpose of this work is to develop methods to ensure a good level of privacy in a classical kind of gossip algorithm, computing an average value.

I.2 Attackers

We will focus attacks against privacy, we are only considering attackers trying to get information from users, not attackers trying to prevent the algorithm to compute a correct result.
Attackers we consider may be of two kinds, possibly both at the same time.
Edge attackers: they are eavesdroppers who can read data exchanged by other peers.
Node attackers: they use peers under their control to get information from other peers.

I.3 Contribution

I.3.a Averaging

Algorithms allowing gossip averaging with privacy protection already exist, but have their limitation. We will propose a new algorithm, with fewer limitations than existing ones.

I.3.b Peer-sampling

Due to the decentralized nature of this kind of algorithm, it requires a lower level protocol, managing the connection between different peers and preventing loss of connectivity, including deliberate isolation of peers by attackers. This kind of protocol is called a peer sampling protocol.
While such protocols already exist, they can be somewhat limited in their capacity to protect efficiently from attacks, especially when we do not want to rely on a central authority for managing the network.
We will develop a new approach of peer sampling, designed to address common limitations of this kind of protocol.

II Previous works

An idea developed in [All+16] is to generate random values following a certain distribution (in their case, Gaussian) and add them to the actual value of the peer to obtain a set of pseudo values of which the mean value is the actual value of the peer. These values were then used as values for a set of pseudo-peers, participating in the gossip averaging algorithm.

II.1 Possible attacks

While encryption provides good protection against an edge attacker, this protocol is still vulnerable to peer attackers in several ways.

II.1.a Get all random values

An obvious attack against this method is to try to get the values of all the pseudo-peers, ideally, before they communicate with anyone else to prevent additional noise from foreign peers. Computing the average of these values will give the attacker the actual value of the peer. The difficulty of this attack has been evaluated by the authors in their paper.

II.1.b Sampling-based estimation

Since the previous attack, which potentially allows the attacker to get the exact value of the peers, is difficult, another, probabilistic, attack can be used. The essential property of pseudo-peers' values is that their average is the actual value of the peer. Even if the attacker can't get all those values, he can still get an estimation of the actual value by taking the average of a subset of values.
We can also see that, while it is a good protection against the first attack, increasing the number of pseudo-peers actually makes this second attack even more efficient, since it will allow the attacker to get a larger set of values, increasing its estimation's precision by the Central Limit Theorem.

II.2 Solution from other authors

While other systems from computing an average value using a gossip protocol may be more resilient to the second kind of attacks, they have there own limitations.
The model presented in [MH13] was proved safe under particular requirements and is designed to work with synchronized peers. The one proposed in [DBR18] requires the random numbers to be generated by a pair of peers, which implies that a peer can't protect its value without another one doing the same thing and have to explicitly give the information that it is protecting its value. In [Lep16], random values are generated locally, but the peers still have to explicitly tell to their peers that the value they send is random.

II.3 Other approaches

Significantly different approaches have also been proposed, based on homomorphic cryptography [Fri10], relying on networks with a controlled topology [Cli+02] [SKM10] or synchronous networks [HMD12].

III A more efficient method

The idea we will introduce in this section also uses random values but in a way that makes sampling-based attacks impossible. Our method will not required synchronization between peers and a peer will be able to protect its value without any of its neighbors doing so and without explicitly informing anyone. In particular, we can have a mix of peers using privacy protection and peers not using it.

III.1 A property of gossip averaging

In this subsection, we will prove that, in a classical gossip averaging algorithm as proposed in [JMB05], if some of the values exchanges by peers are replaced by random values and, if later, an appropriate correction is done on the value of peers, then, the algorithm will converge the same way as without these operations. Let's consider a classical theoretical continuous-time model of gossip averaging where each peer is a vertex of a (connected) graph $G(V, E)$.

Theorem (Correctness Theorem)**.** *If the values $x_0(t), ..., x_k(t)$ of all vertices of any subset $P \subseteq V$ ($|P| = k$) are replaced $r_0; ...; r_k$ successive times by the values $y_{0,0}; ...; y_{0,r_0}; ...y_{k,0}; ...; y_{k,r_k}$ at times $t_{0,0}; ...; t_{0,r_0}; ...; t_{k,0}; ...; t_{k,r_k}$ ($x_i(t_{i,j} + \varepsilon) = y_{i,j}$) and later (say at time t_*), we add them the values $\sum_{i=0}^{r_0}(x_0(t_{0,i}) - y_{0,i}); ...; \sum_{i=0}^{r_k}(x_k(t_{k,i}) - y_{k,i})$ ($\forall_{0 \leqslant i \leqslant k} x_i(t_* + \varepsilon) = x_i(t_*) + \sum_{j=0}^{r_0}(x_0(t_{0,j}) - y_{0,j})$), then, executing a classical gossip averaging algorithm on this graph will ultimately make every vertex's value converge to the average of all initial values.*

To prove this theorem, we will start with this simple lemma:

Lemma (Local Correctness Lemma)**.** *If, at time t, we replace the value $x(t)$ of a vertex by y ($x_i(t+\varepsilon) = y$) and then, at time t', we add to $x(t')$ the value $x(t) - y$ ($x_i(t + \varepsilon) = x_i(t') + x(t) - y$), then, the average value of all vertices in the graph after the transformation at time t' is the same as before the transformation at time t. $x(t) - y$*

Proof. In the transformation at time t, we removed $x(t) - y$ to the value of the vertex we modified. Averaging operations having no effect on the average value of all vertices, if, at time t', we add $x(t) - y$ to the value of any vertex, the original average value of all vertices is restored.

Now, we can demonstrate our Correctness Theorem.

Proof. We just apply our Local Correctness Lemma to all $y_{v,i}$ for every vertex of $v \in P$.

Now we know that we can replace an arbitrary number of times the value of a peer by a random value and the averaging algorithm will still work, provided that we later add to the value of our peer the sum of all the differences between the value our peer had at the time of replacement and the random value.

III.2 A new algorithm...

Let's introduce the following algorithm. Each time a peer performs a value exchange with another, it will replace its actual value with a random value (with any non-constant distribution), until it has sent a certain number of random values, and then, add to its value the above mentioned sum of differences. When this is done, the peer will execute a classical gossip averaging algorithm. We assume that all communications are encrypted and that an eavesdropper may only know the time, the sender and the receiver of any communication, not its content. This process is described in the following algorithm (function answer is called upon receiving a message):

Algorithm Private Gossip Averaging

```
1: function privateGossipAverage(val, privLvl)
2:     err ← 0
3:     for i ← 0 to privLvl − 1 do
4:         peer ← randomPeer()
5:         fakeVal ← rand()
6:         err+ = val − fakeVal
7:         sendTo(peer, fakeVal)
8:         rcv ← recvFrom(peer)
9:         val ← (fakeVal+rcv)/2
10:    end for
11:    val+ = err
12:    loop
13:        peer ← randomPeer()
14:        sendTo(peer, val)
15:        rcv ← recvFrom(peer)
16:        val ← (val+rcv)/2
17:    end loop
18: end function
19: function answer
20:    rcv ← recvFrom(peer)
21:    if i < privLvl then
22:        fakeVal ← rand()
23:        err+ = val − fakeVal
24:        sendTo(peer, fakeVal)
25:        val ← (fakeVal+rcv)/2
26:    else
27:        sendTo(peer, val)
28:        val ← (val+rcv)/2
29:    end if
30: end function
```

III.2.a Example

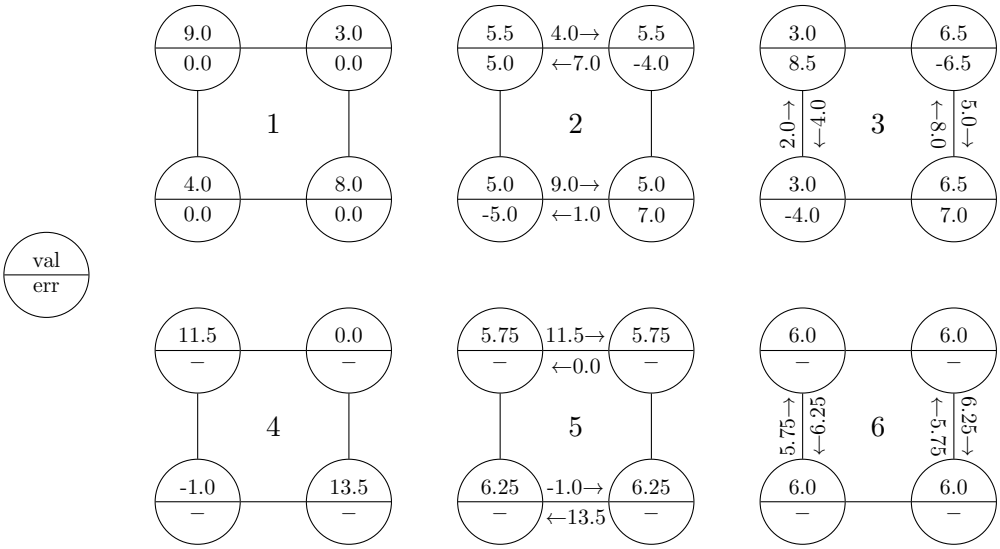

Figure 1: 4 nodes executing our algorithm

III.3 ...with good properties

We first notice that this process does not change the average value of the whole set of nodes.

Property (Correctness property)**.** *If the Gossip Privacy Protector algorithm is applied by any number of nodes at the start of a classical gossip averaging algorithm, this does not change the value to which all nodes will converge.*

This is a simple application of our Correctness Theorem.
We will now prove that this algorithm has good properties for privacy preservation.

Property (Values' Independence Property)**.** *All random values sent are independent from each other and from the original value of the peer.*

By definition (of our algorithm).

III.4 Direct Attacks

For the following properties, we will assume that the attacker can only get information directly from its target peer, the indirect case will be covered later.

Property (Deterministic Safety Property)**.** *An attacker needs to get all the random values sent to compute the exact original value of the peer.*

Proof. We will use the same notation as in the algorithm (with time indexes added, and p instead of *privLvl*). The first non-random value is exactly:

$$val_p = val_{p-1} + err_{p-1} = \frac{fakeVal_{p-1} + rcv_{p-1}}{2} + \sum_{i=0}^{p-1}(val_i - fakeVal_i)$$

$$val_p = \frac{fakeVal_{p-1} + rcv_{p-1}}{2} + \sum_{i=1}^{p-1}\left(\frac{fakeVal_{i-1} + rcv_{i-1}}{2} - fakeVal_i\right) + val_0 - fakeVal_0$$

$$val_p = \sum_{i=0}^{p-1}\left(\frac{rcv_i - fakeVal_i}{2}\right) + val_0$$

So:

$$val_0 = val_p - \sum_{i=0}^{p-1} \left(\frac{rcv_i - fakeVal_i}{2} \right)$$

We see that knowledge of all random values (plus the associated answers from contacted peers) is required to compute the original value from the first non-random value exchanged. All later non-random values having even more noise from other peers and no non-random values being transmitted before, this proves the property.

Property (Probabilistic Safety Property)**.** *If the attacker lacks k random values (but not necessarily the associated answers), the best he can do is to take the expected value for them (assuming he knows it), with a level of uncertainty at least as great as if the value he had to guess was $\frac{\sum_{i=0}^{k} fakeVal_i}{2}$.*

Proof. Due to the Values' Independence Property, it is impossible to guess anything from these values, except using their relation to val_p. Since $val_0 = val_p - \sum_{i=0}^{p-1} \left(\frac{rcv_i - fakeVal_i}{2} \right)$, if the attacker lacks k values, he has to guess them without any hint, and those values are independent, so, the best guess he can do is taking their expected values and nothing will give him a lower level of uncertainty than the natural level of uncertainty of those random variables.

From the Deterministic Safety Property, we can derive that, if the peer sampling is effectively random, the chance that an attacker will get the exact original value of a peer is $\leqslant \tau^p$, were $\tau = \frac{\#\text{corrupted peers}}{\#\text{peers}}$.

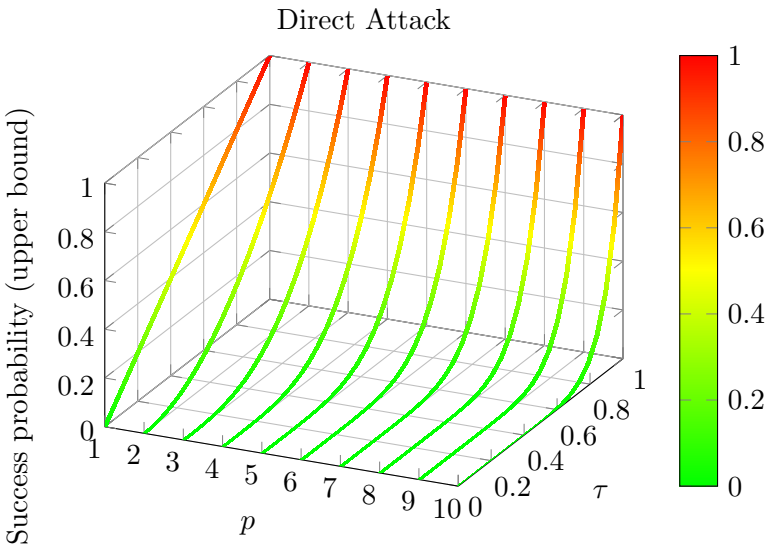

Figure 2: Probability of success of a Direct attack (upper bound)

III.5 Indirect Attacks

Indirect attacks are only possible if the attacker is both a peer and an edge attacker, since it requires fake peers to get values and eavesdroppers to know which values he needs. Also note that the attacker needs to get values older than the ones he wants to obtain. Also, they will only work if all involved peers, but the target, are no more in the privacy generation phase of the algorithm, since values sent by a peer generating privacy are all unrelated to each other (Values' Independence Property) and it will be impossible for any observer to know exactly what a peer got from which other peer in this phase. This implies that, if all peers start the process at the same time, it's unlikely that these attacks will be possible for nearly all random values exchanged. Except for the very simple case of spying target peer's neighbors, this kind of attack will probably be very impractical for most, if not

all, cases on a real network.

These attacks rely one the following principle: if peer i exchanges values with peer j at time t, it's possible to compute the values exchanged if you know $v_j(t)$ (the value sent by j) and $v_j(t+1) = \frac{v_i(t)+v_j(t)}{2}$.

$$v_i(t) = 2\frac{v_i(t)+v_j(t)}{2} - v_j(t) = 2v_j(t+1) - v_j(t)$$

This attack can be iterated, if the attacker lacks one of, or both, $v_j(t)$ and $v_j(t+1)$, you may get it using the same attack. We call this Higher Order Indirect Attacks, the base case being First Order Indirect Attacks.

III.5.a First Order Indirect Attacks

First order indirect attacks give the attacker a second chance to get a value, but require two contacts instead of one. This increases the upper bound of the probability of an exact evaluation from $\leqslant \tau^p$ to $\leqslant (\tau + (1-\tau)\tau^2))^p = \leqslant (\tau + \tau^2 - \tau^3))^p$. We will explain later how we ensure that all contacts a peer have are with random peers, even when the peer is contacted.

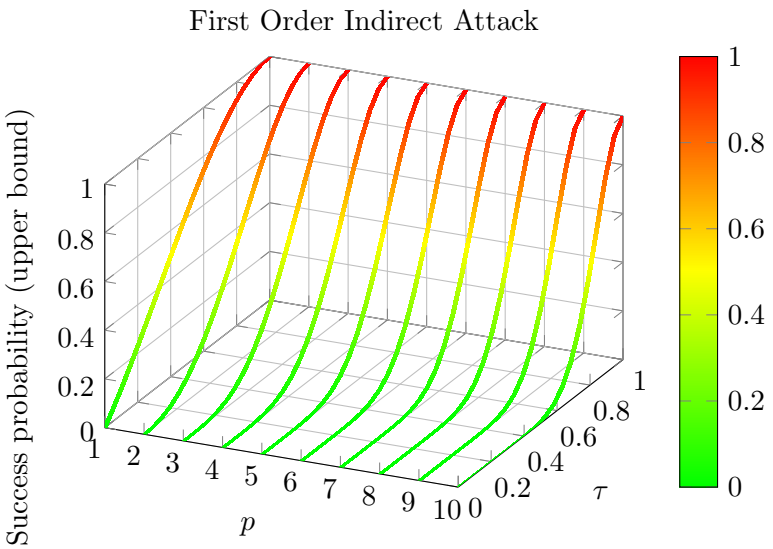

Figure 3: Probability of success of a First Order Indirect attack (upper bound)

III.5.b Higher Order Indirect Attacks

While higher order indirect attacks are very unlikely in practice, from a theoretical point of view, they can be very powerful. We will prove two results showing the limitations of these attacks.

The analyze we are performing have no time nor space limits; the attacker may take arbitrary long (finite) time to get the information he wants and may have to use arbitrary old information. We assume here that our attacker is an universal eavesdropper (he can see all messages). We also assume that, of all the exchanges considered, there is no pair of consecutive exchanges between the same peers, neither pair of exchanges with one (or both) peer in common separated by only one exchange. These last assertions are true with uniform random peer sampling over an infinite set of peers.

Theorem (Higher Order Survival Theorem)**.** *If $\tau < 1/2$, the attacker will never get the information he wants with probability $\geqslant 1 - \frac{1-2\tau(1-\tau)-\sqrt{1-4\tau(1-\tau)}}{2(1-\tau)^2}$.*

Proof. We model the tree of exchanges induced by the higher order attack by a Galton–Watson process.

The number of descendants of an individual is the number of exchanges made just before and after the exchange corresponding to the individual which are not made with a corrupted peer. This gives the following probabilities: $p_0 = \tau^2$, $p_1 = 2\tau(1-\tau)$, $p_2 = (1-\tau)^2$. The average number of descendants is $m = p_1 + 2p_2 = 2\tau(1-\tau) + 2(1-\tau)^2$.
Let's analyze the branching process' behavior as a function of τ.

$$m = 2\tau(1-\tau) + 2(1-\tau)^2 = 2\tau - 2\tau^2 + 2 - 4\tau + 2\tau^2 = 2 - 2\tau$$

So $m = 1 \Leftrightarrow 2\tau = 1 \Leftrightarrow \tau = 1/2$. Since $p_1 = 2\tau(1-\tau) = 2{1/2}^2 = 1/2 < 1$, the probability of extinction in the critical case is 1.
For the super-critical case, $m > 1$, let's analyze the generating function.

$$\varphi(s) = \sum_{n \geqslant 0} p_n s^n = p_2 s^2 + p_1 s + p_0 = (1-\tau)^2 s^2 + 2\tau(1-\tau)s + \tau^2$$

We want $\varphi(s) = s$, we search the roots of the polynomial $\varphi(s) - s$.

$$\varphi(s) - s = (1-\tau)^2 s^2 + (2\tau(1-\tau) - 1)s + \tau^2$$
$$\Delta = (2\tau(1-\tau) - 1)^2 - 4\tau^2(1-\tau)^2$$
$$\Delta = 4\tau^2(1-\tau)^2 - 4\tau(1-\tau) + 1 - 4\tau^2(1-\tau)^2$$
$$\Delta = 1 - 4\tau(1-\tau)$$

For $0 \leqslant \tau < 1/2$, $\Delta > 0$.

$$r = \frac{1 - 2\tau(1-\tau) \pm \sqrt{1 - 4\tau(1-\tau)}}{2(1-\tau)^2}$$

Of the two solutions, the lower one living in $[0, 1]$ is always $r = \frac{1-2\tau(1-\tau)-\sqrt{1-4\tau(1-\tau)}}{2(1-\tau)^2}$ (the other being always 1 for $0 \leqslant \tau < 1/2$).
The probability of extinction in the super-critical case is $\frac{1-2\tau(1-\tau)-\sqrt{1-4\tau(1-\tau)}}{2(1-\tau)^2}$, so, the probability of survival is: $1 - \frac{1-2\tau(1-\tau)-\sqrt{1-4\tau(1-\tau)}}{2(1-\tau)^2}$.
□

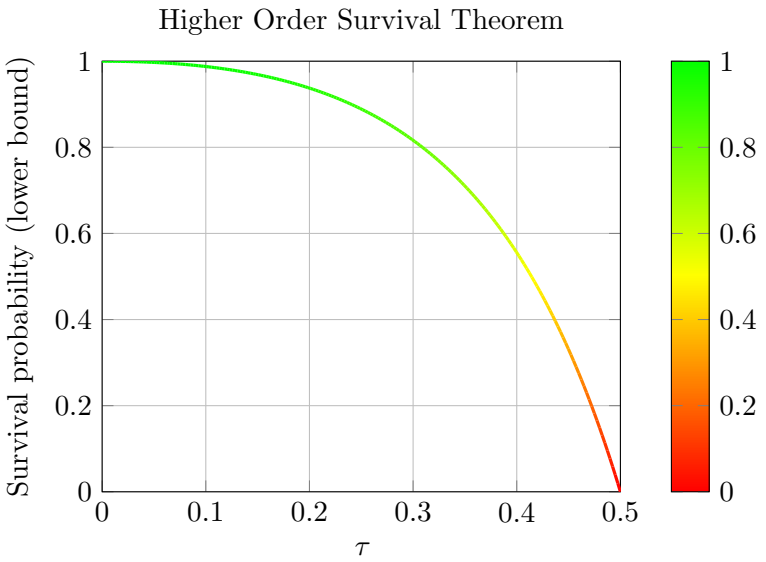

Figure 4: Survival probability (lower bound) given by our Higher Order Survival Theorem

Since our first theorem is subject a major assumption (not twice the same peer) and doesn't give us anything for $\tau \geqslant 1/2$, we will prove another theorem, which doesn't have those limitations but is not applicable to universal eavesdroppers.
For this theorem, we introduce the value $\theta = \frac{\#unsafeedges}{\#edges}$ in the sub-graph from which all corrupted peers have been removed. We assume $0 < \tau < 1$ (if not, there is either no attacker or the attacker controls the whole network) and $0 < \theta < 1$ (if not, we have either an universal eavesdropper or no eavesdropper at all).

Theorem (Higher Order Escape Theorem)**.** *There is a* $\geqslant 1 - \frac{\tau}{1-\theta(1-\tau)}$ *probability that the attacker will never get the value of an exchange.*

Proof. This theorem relies on the fact that, if, at some time, a communication is made via a safe edge (which the eavesdropper can't spy), then, the attacker will never know which peer to spy for the next step of his indirect attack. Since we do not assume that the same peer won't be randomly selected several times and we only want a lower bound on the probability of success of an attack, we will consider a worst case, where only one new exchange needs to be captured at each step.
For the first step, the probability of the exchange not being captured is $1-\tau$ and then, it's probability of happening on a safe edge is $1-\theta$, which gives a $(1-\tau)(1-\theta)$ probability that the attacker will never learn the value of the exchange. If the communication happens on an unsafe edge, then, at each step, the chance of reaching a safe edge is multiplied by $\theta(1-\tau)$ (the previous edge was not safe and the next peer is not corrupted). So, at step k (assuming that the first step is 0), the probability of not having reached an corrupted peers and reaching a safe edge for the first time at this step is $\theta^k(1-\tau)^{k+1}(1-\theta)$. This is a geometric series, so we can compute its sum.

$$E_n = \sum_{k=0}^{n} \theta^k (1-\tau)^{k+1}(1-\theta)$$

$$E_n = (1-\tau)(1-\theta)\frac{1-\theta^{k+1}(1-\tau)^{k+1}}{1-\theta(1-\tau)}$$

Since $\theta(1-\tau) < 1$

$$E_{+\infty} = (1-\tau)(1-\theta)\frac{1}{1-\theta(1-\tau)} = \frac{(1-\tau)(1-\theta)}{1-\theta(1-\tau)}$$

$$E_{+\infty} = \frac{1-\theta-\tau+\theta\tau}{1-\theta+\theta\tau} = 1 - \frac{\tau}{1-\theta+\theta\tau}$$

$$E_{+\infty} = 1 - \frac{\tau}{1-\theta(1-\tau)}$$

□

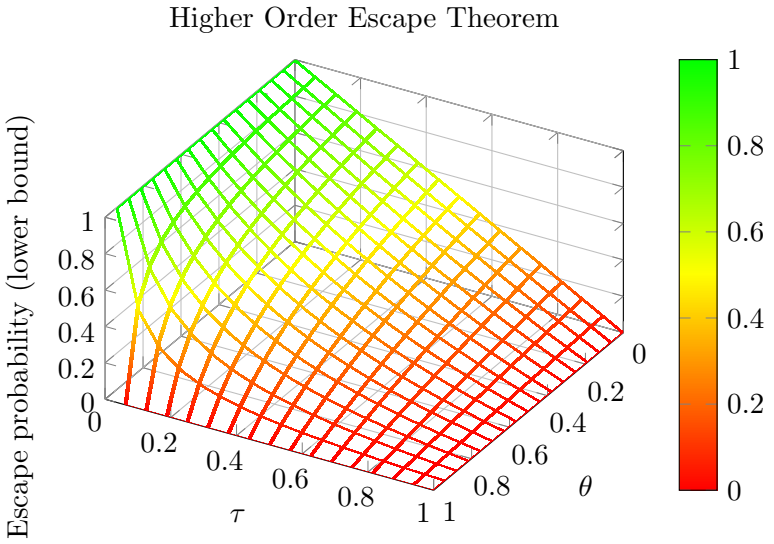

Figure 5: Escape probability (lower bound) given by our Higher Order Escape Theorem

III.5.c Getting random contacts

The method we propose to ensure uniform randomness in the contacts we get is to never answer directly to a contacting peer but rather wait for another peer to ask for forwarding and forward him the information received from the contacting peer, then reply with the answer from the other peer.
This way, when a peer actually answers to someone, the peer has selected a random peer on its side before, so it is not possible to try to get contact specifically with it, since the contact forwarded has been established before the forwarding was proposed. A system of timeouts or timestamps and synchronized clocks can be used to ensure it is not possible to abuse of a lower ping to send a contact request during the transmission time of the forward request.
Another point is that it is not possible to know who actually answered a contact, so, this makes it more difficult for attackers to know past values of peers, which is required for Indirect Attacks.
A system of resource balancing is also required to ensure that no peers communicates more than other ones, the attacker may still use more fake peers to have more communications through.

Now we know that our algorithm does not affect the final result, that it is difficult to get the exact value of a peer and that trying to guess this value will leave the attacker with a great level of uncertainty.

IV Peer Sampling

For our algorithm to work properly, we need a good peer sampling algorithm.
Previous works in this field were either ignorant of the issue of Byzantine attacks[Néd+17] or required access to the network to be under control of a trusted authority[Bor+09] to prevent Sybil attacks[Dou02].

In the second part of this work, we will present a new approach of gossip peer sampling: *Hierarchically Addressed Peer Sampling* (*HAPS*).

IV.1 Hierarchically Addressed Peer Sampling

The main idea for this kind of peer sampling is that we do not assume that peer ids are just random integers. We now consider peer ids are addresses, attributed hierarchically, like IP addresses.

Here, a peer id can be considered equivalent to the pair formed by an IP address and a port number. Since gossip protocols are likely to be used on the Internet network, or similarly working networks, using this kind of hierarchical id attribution seemed interesting to us, as it will not require any central authority, other than those that are required for the Internet to work.

If an attacker wants to create multiple peers, those peers' addresses will only differ by their last bits if he is operating from one unique computer, local network or server. To get significantly different addresses, the attacker will need either an addresses bloc with a small mask or multiple addresses spread in the whole network. The first solution will have a cost due to the expensive nature of big addresses blocs, the second will require either a botnet, multiple attackers or to pay for multiple addresses.
Protection from address spoofing can be achieved simply by always pinging a peer before adding it to the local view. Getting an answer to this ping will ensure that the address added to the local is effectively a peer executing the protocol, preventing an attack based on the injection of random addresses to isolate a peer. Injecting addresses of real peers not under its control is useless for an attacker.

Based on these specificity of hierarchical addressing, we propose a peer sampling protocol designed to produce local views containing peers spread over the whole address plan, even if the address received from peers are not, making it more difficult (expensive) for an attacker to flood an user's view.
To achieve this, we replace the set container used to store peers in other peer sampling algorithms by a new container called *Buffered Probabilistic Binary Addresses Tree* (*BPBA-Tree*), that we will define in the following subsection.
This container makes peers too close on the address plan random peers, having a < 1 probability of being part of the peer's view. This way, even if a peer's local view gets polluted by an important number of peers from a unique attacker, with close addresses, the probability for these peers to be selected randomly of even considered part of the peers view will remain low.
In addition, some peers, even closer, will be considered temporary; a clean function, to be called periodically, will remove those peers. This will allow a peer to keep its view of reasonable size with a very low risk of removing all (or most) good nodes from it, since it will only remove nodes that are too close to each other. Compared to immediate removal, this periodical cleaning makes it more difficult for an attacker to remove a specific peer from a local view, since he will not be able to try adding close addresses to the tree until the targeted address is removed, he will only have a fixed chance of success at each cleaning.
Using this new container, peer sampling can be done with a protocol similar to the one described in[ADH05], where peers send requests to others and then, after getting added to their local views, receive answers consisting of a sample of their local views. The main difference is that the local view's size is not strictly fixed and that a clean operation will have to be executed periodically.

The following algorithm describes the protocol. Function "answer" is called upon receiving a request from another peer.

Algorithm Hierarchically Addressed Peer Sampling

```
1: function hierarchicallyAddressedPeerSampling(requestEvery, cleanEvery, pullSize)
2:     loop
3:         for i ← 1 to cleanEvery do
4:             peer ← Tree.rand()
5:             sendTo(peer, pullSize)
6:             sample ← recvFrom(peer)
7:             for all addr ∈ sample do
8:                 if ping(addr) then
9:                     Tree.insert(addr)
10:                end if
11:            end for
12:            idle(requestEvery)
13:        end for
14:        Tree.clean()
15:    end loop
16: end function
17: function answer
18:    if ping(addr) then
19:        size ← recvFrom(peer)
20:        sample ← Tree.randSet(size)
21:        sendTo(peer, sample)
22:        Tree.insert(peer)
23:    end if
24: end function
```

IV.2 Buffered Probabilistic Binary Addresses Tree

Our structure is a (binary) tree where nodes represent either an address or a subnet. All nodes of the tree have two values: an address (subnet) and a (subnet) mask. Addresses (of peers) are stored as leaves of a binary tree. Non-leaf nodes in the tree corresponds to the common prefix of their two children.

The container has two special parameters: a *deterministic threshold* an a *keep threshold*. All nodes descending from a node with a mask >=deterministic threshold are *random nodes*, all nodes descending from a node with a mask >=keep threshold are *temporary nodes* (in particular, they are random nodes). Nodes being leaves for the subtree expurgated of random (resp. temporary) nodes are called *deterministic leaves* (resp. *keep leaves*).
The particularity of random nodes is that they are not always considered to be part of the tree, instead, their probability of being considered part of the tree is $\frac{1}{2^d}$ where d is the depth of the node relatively to the last non-random node in the path from the root to them.
For temporary nodes, they can, in addition, be removed during a specific *clean* operation performed on the tree. Their probability of being removed during such an operation is $1 - \frac{1}{2^d}$ where d is the depth of the node relatively to the last non-temporary node in the path from the root to them.

When looking for an address (traversing the tree from its root), once a deterministic leaf is reached, each change of node will have a 1/2 probability to stop the search with a false return. Similarly, when a random peer is requested, a deterministic leaf will be chosen uniformly at random, then the tree will be traversed, choosing a random child at each node until a leaf is reached.

When the clean function is called, each keep leaf (if it is not already a leaf) will be replaced by a

leaf of the subtree under it. This leaf is chosen the same way as when selecting a random address.

If an attacker tries to flood a peer's local view with addresses from a subnet with a mask longer than the deterministic threshold, the sum of the probabilities of existence of all these addresses will be the same as if only one was added. The same applies for the probability of not being removed by a clean operation.
This ways, attacks relying on a high number of nodes from a small (relatively to the thresholds) subnet are completely ineffective.

IV.2.a Definition of classes and structures

class **Tree** (private values)	
Node*	root
UInt	size
UInt	addrLen
UInt	deterThld
UInt	keepThld
Set<Node*>	deterLeaves
Set<Node*>	keepLeaves

struct **Node**	
Address	subnet
UInt	mask
Node*	parent
Node*	zero
Node*	one

class **Tree** (public functions)	
←UInt ←UInt ←UInt	Tree(addrBits,deterLimit,keepLimit)
Void ←Address	insert(address)
Void ←Address	remove(address)
Bool ←Address	find(address)
Bool ←Address	findProba(address)
Address ←Void	rand()
Set<Address> ←UInt	randSet(s)
Void ←Void	clean()
Void ←UInt	setDeterThld(deterLimit)
Void ←UInt	setKeepThld(keepLimit)
UInt ←Void	getAddrLen()
UInt ←Void	getDeterThld()
UInt ←Void	getKeepThld()
UInt ←Void	getSize()
UInt ←Void	getDeterSize()
UInt ←Void	getKeepSize()
Bool ←Void	isEmpty()

IV.2.b Short description of important functions

Algorithm Buffered Probabilistic Binary Addresses Tree (insert)

1: **function** Tree.insert(*address*)
2: find address' node's futur parent
3: build a copy of parent node
4: build new node
5: reduce parent node's mask to the length of the longest commun prefix
6: update keep leaves and deter leaves sets
7: insert node and copy of parent node as children of parent node
8: **end function**

Algorithm Buffered Probabilistic Binary Addresses Tree (remove)

```
1: function Tree.remove(address)
2:     search address' node's position
3:     if found then
4:         remove node
5:         replace node's parent by its remaining child
6:         update keep leaves and deter leaves sets
7:     end if
8: end function
```

Algorithm Buffered Probabilistic Binary Addresses Tree (search)

```
1: function Tree.find(address)
2:     search address' node's position
3:     if found then
4:         return ⊤
5:     else
6:         return ⊥
7:     end if
8: end function
9: function Tree.findProba(address)
10:    search address' node's position until hitting a node with a mask > deterministic threshold
11:    continue searching but, at each steep, abort with probability 1/2
12:    if found then
13:        return ⊤
14:    else
15:        return ⊥
16:    end if
17: end function
```

Algorithm Buffered Probabilistic Binary Addresses Tree (rand)

```
1: function Tree.rand
2:     chose a deterministic leaf randomly
3:     descend the tree, choosing a random child at each step, until reaching a leaf
4:     return leaf's address
5: end function
6: function Tree.randSet(s)
7:     chose a set of deterministic leaves randomly
8:     for all element of the set do
9:         descend the tree, choosing a random child at each step, until reaching a leaf
10:    end for
11:    return leaves' addresses
12: end function
```

Algorithm Buffered Probabilistic Binary Addresses Tree (clean)

```
1: function Tree.clean
2:     for all keep leaf do
3:         descend the tree, choosing a random child at each step, until reaching a leaf
4:         replace the keep leaf by the leaf
5:     end for
6: end function
```

IV.2.c Example of tree

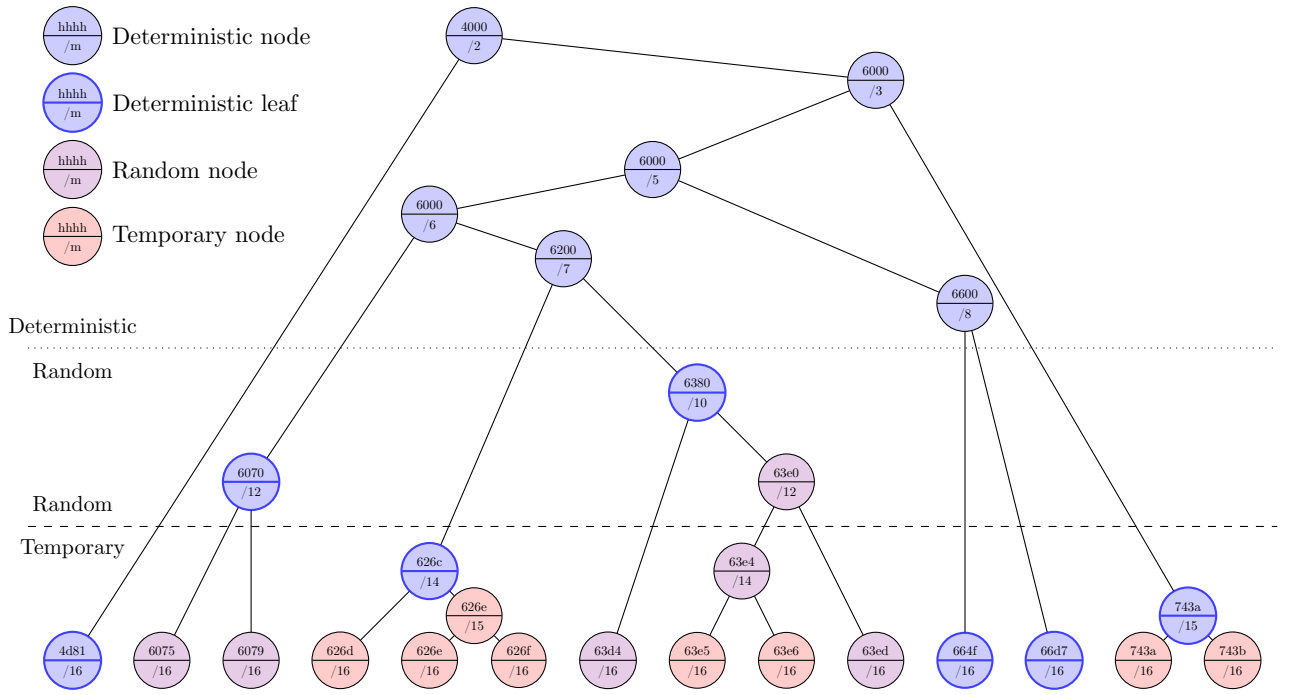

Figure 6: Example of Buffered Probabilistic Binary Addresses Tree

IV.2.d Examples of operations

On the example tree, when calling the *FindProba* function with *6075* as parameter, we will descend the tree until reaching a deterministic leaf: 4000/2→6000/3→6000/5→6000/6→6070/12. Then, with a $1/2$ probability, we will hit the 6075/16 node and return True, else, we will stop there and return false.

When calling the *Rand* function, we will choose randomly (uniformly) a deterministic leaf, let us say it is 6380/10. Then, we will descend the tree, choosing a child randomly at each node until reaching a leaf. For example, we may do 6380/10→63e0/12→63ed/16, with a $1/2^2 = 1/4$ probability.

When executing the *clean* procedure, for each keep leaf, we will descend the tree, choosing a child randomly at each node until reaching a leaf and replace the keep leaf by the leaf. For example 63e4/14 may be replaced by 63e6/16 with a $1/2$ probability.

IV.2.e Complete definition of non-trivial functions

In the following pseudo-code, $\oplus$ is the bitwise XOR, $\ll$ is the left-shift operator, $\gg$ is the right-shift operator (for unsigned values) and $\neg$, $\vee$ and $\wedge$ are, respectively, the logical (not bitwise) NOT, OR and AND (with any non-0 value being True).

Algorithm Buffered Probabilistic Binary Addresses Tree (constructor)

1: **function** Tree($addrBits, keepLimit, deterLimit$)
2: $addrLen \leftarrow addrBits$
3: $keepThld \leftarrow keepLimit$
4: $deterThld \leftarrow deterLimit$
5: $size \leftarrow 0$
6: $deterLeaves \leftarrow$ Set.emptySet()
7: $keepLeaves \leftarrow$ Set.emptySet()
8: **end function**

Algorithm Buffered Probabilistic Binary Addresses Tree (insert)

```
1: function TREE.INSERT(address)
2:     if size = 0 then
3:         root ← NODE(address, addrLen, NULL, NULL, NULL)
4:         size ← 1
5:         deterLeaves.insert(root)
6:         keepLeaves.insert(root)
7:     else
8:         node ← root
9:         while ¬((address ⊕ node.subnet) ≫ (addrLen − node.mask)) do
10:            if node.mask = addrLen then
11:                return
12:            end if
13:            if (address ≪ node.mask) ≫ (addrLen − 1) then
14:                node ← node.one
15:            else
16:                node ← node.zero
17:            end if
18:        end while
19:        splitNode ← NODE(node.subnet, node.mask, node, node.zero, node.one)
20:        newNode ← NODE(address, addrLen, node, NULL, NULL)
21:        size + +
22:        if node.mask < addrLen then
23:            node.zero.parent ← splitNode
24:            node.one.parent ← splitNode
25:        end if
26:        node.mask ← cntLeadZeros(address ⊕ node.subnet)
27:        if node.mask < keepThld then
28:            if splitNode.mask >= keepThld then
29:                keepLeaves.remove(node)
30:                keepLeaves.insert(splitNode)
31:            end if
32:            keepLeaves.insert(newNode)
33:            if node.mask < deterThld then
34:                if splitNode.mask >= deterThld then
35:                    deterLeaves.remove(node)
36:                    deterLeaves.insert(splitNode)
37:                end if
38:                deterLeaves.insert(newNode)
39:            end if
40:        end if
41:        if address[addrLen − node.mask − 1] then
42:            node.zero ← splitNode
43:            node.one ← newNode
44:        else
45:            node.zero ← newNode
46:            node.one ← splitNode
47:        end if
48:    end if
49: end function
```

Algorithm Buffered Probabilistic Binary Addresses Tree (remove)

```
1: function TREE.REMOVE(address)
2:     if size = 1 ∧ addr = root.subnet then
3:         size ← 0
4:         DETERLEAVES.REMOVE(root)
5:         KEEPLEAVES.REMOVE(root)
6:     else if size > 1 then
7:         node ← root
8:         if ¬((address ⊕ node.subnet) ≫ (addrLen − node.mask)) then
9:             while node.mask < addrLen do
10:                if (address ≪ node.mask) ≫ (addrLen − 1) then
11:                    node ← node.one
12:                else
13:                    node ← node.zero
14:                end if
15:                if (address ⊕ node.subnet) ≫ (addrLen − node.mask) then
16:                    return
17:                end if
18:            end while
19:            size − −
20:            deleteNode ← node
21:            if node == node.parent.zero then
22:                fusionNode ← node.parent.one
23:            else
24:                fusionNode ← node.parent.zero
25:            end if
26:            node ← node.parent
27:            if node.mask < keepThld then
28:                KEEPLEAVES.REMOVE(deleteNode)
29:                if fusionNode.mask >= keepThld then
30:                    KEEPLEAVES.REMOVE(fusionNode)
31:                    KEEPLEAVES.INSERT(node)
32:                end if
33:                if node.mask < deterThld then
34:                    DETERLEAVES.REMOVE(deleteNode)
35:                    if fusionNode.mask >= deterThld then
36:                        DETERLEAVES.REMOVE(fusionNode)
37:                        DETERLEAVES.INSERT(node)
38:                    end if
39:                end if
40:            end if
41:            node.subnet ← fusionNode.subnet
42:            node.mask ← fusionNode.mask
43:            if node.mask < addrLen then
44:                fusionNode.zero.parent ← node
45:                fusionNode.one.parent ← node
46:            end if
47:            node.zero ← fusionNode.zero
48:            node.one ← fusionNode.one
49:        end if
50:    end if
51: end function
```

Algorithm Buffered Probabilistic Binary Addresses Tree (search)

```
1: function TREE.FIND(address)
2:     if size = 0 ∨ ((address ⊕ root.subnet) ≫ (addrLen − root.mask)) then
3:         return ⊥
4:     end if
5:     node ← root
6:     while node.mask < addrLen do
7:         if (address ≪ node.mask) ≫ (addrLen − 1) then
8:             node ← node.one
9:         else
10:            node ← node.zero
11:        end if
12:        if (address ⊕ node.subnet) ≫ (addrLen − node.mask) then
13:            return ⊥
14:        end if
15:    end while
16:    return ⊤
17: end function
18: function TREE.FINDPROBA(address)
19:    if size = 0 ∨ ((address ⊕ root.subnet) ≫ (addrLen − root.mask)) then
20:        return ⊥
21:    end if
22:    node ← root
23:    while node.mask < addrLen do
24:        if (node.mask => deterThld)∧ RAND(⊥, ⊤) then
25:            return ⊥
26:        end if
27:        if (address ≪ node.mask) ≫ (addrLen − 1) then
28:            node ← node.one
29:        else
30:            node ← node.zero
31:        end if
32:        if (address ⊕ node.subnet) ≫ (addrLen − node.mask) then
33:            return ⊥
34:        end if
35:    end while
36:    return ⊤
37: end function
```

Algorithm Buffered Probabilistic Binary Addresses Tree (rand)

```
1: function Tree.rand
2:     node ← rand(deterLeaves)
3:     while node.mask < addrLen do
4:         node ← rand(node.zero, node.one)
5:     end while
6:     return node.subnet
7: end function
8: function Tree.randSet(s)
9:     nodeSet ← rand(deterLeaves, s)
10:    addrSet ← ∅
11:    for all node ∈ nodeSet do
12:        while node.mask < addrLen do
13:            node ← rand(node.zero, node.one)
14:        end while
15:        AddrSet.insert(node.subnet)
16:    end for
17:    return AddrSet
18: end function
```

Algorithm Buffered Probabilistic Binary Addresses Tree (clean)

```
1: function Tree.clean
2:     for all base ∈ keepLeaves do
3:         if base.mask < addrLen then
4:             node ← base
5:             repeat
6:                 if rand(⊥, ⊤) then
7:                     size− = countLeaves(node.zero)
8:                     node ← node.one
9:                 else
10:                    size− = countLeaves(node.one)
11:                    node ← node.zero
12:                end if
13:            until node.mask = addrLen
14:            base.subnet ← node.subnet
15:            base.mask ← addrLen
16:        end if
17:    end for
18: end function
```

Algorithm Buffered Probabilistic Binary Addresses Tree (change thresholds)

```
1: function Tree.setDeter(deterLimit)
2:     if deterLimit > deterThld then
3:         for all node ∈ deterLeaves do
4:             if node.mask < deterLimit then
5:                 deterLeaves.remove(node)
6:                 deterLeaves.union(getLeaves(node.zero, limit))
7:                 deterLeaves.union(getLeaves(node.one, limit))
8:             end if
9:         end for
10:    else if deterLimit < deterThld then
11:        for all node ∈ deterLeaves do
12:            if node.parent ≠ NULL ∧ node.parent.mask >= deterLimit then
13:                deterLeaves.remove(node)
14:                repeat
15:                    node ← node.parent
16:                until node.parent = NULL ∨ node.parent.mask < deterLimit
17:                deterLeaves.insert(node)
18:            end if
19:        end for
20:    end if
21:    deterThld ← deterLimit
22: end function
23: function Tree.setKeep(keepLimit)
24:    if keepLimit > keepThld then
25:        for all node ∈ keepLeaves do
26:            if node.mask < keepLimit then
27:                keepLeaves.remove(node)
28:                keepLeaves.union(getLeaves(node.zero, limit))
29:                keepLeaves.union(getLeaves(node.one, limit))
30:            end if
31:        end for
32:    else if keepLimit < keepThld then
33:        for all node ∈ keepLeaves do
34:            if node.parent ≠ NULL ∧ node.parent.mask >= keepLimit then
35:                keepLeaves.remove(node)
36:                repeat
37:                    node ← node.parent
38:                until node.parent = NULL ∨ node.parent.mask < keepLimit
39:                keepLeaves.insert(node)
40:            end if
41:        end for
42:    end if
43:    keepThld ← keepLimit
44: end function
45: function Tree.getLeavesRec(private)(node, limit)
46:    ans ← ∅
47:    if node.mask >= limit then
48:        return Set(node)
49:    else
50:        return Set.union(getLeaves(node.zero, limit),getLeaves(node.one, limit))
51:    end if
52: end function
```

V Experimentation

Evaluating a peer sampling protocol like ours by a theoretical analysis being complicated, we evaluated it experimentally. Our tests are divided in two parts: first, an analysis of the properties of the graph induced by the structure of the resulting network, second, an comparative evaluation of the performances of our averaging algorithm when executed in association with our peer sampling protocol, the performances obtained when used with a Req-Pull peer-sampling, similar to HAPS except that it is not using a BPBA-Tree (and, thus, do not have the safety properties associated with hierarchical addressing) but a fixed-size queue of peers and the performances obtained when used with a perfect peer sampling protocol (complete graph).

V.1 Analysis of the graph

Since, not only a peer a may be part of the local view of a peer b without b being in the local view of a, but also, our BPBA-Tree container gives peers a $\notin \{0, 1\}$ probability of being part of a local view, the graphs we are going to process for this analysis will be directed weighted graphs. In these graphs, there is an edge from a to b if and only if b is part of a's local view and the weight of this edge is the probability of b to be considered part of the local view of a when executing a $FindProba$ request on a's BPBA-Tree container.

V.1.a Vertices' degrees

Here, for every vertex, we compute separately the number of edges to it and from it. We then sort these numbers in increasing order and plot the result.
For this test, we used a deterministic threshold of 4, a keep threshold of 6 with 16 bits addresses and 1000 peers. The result is the average over 10 runs.

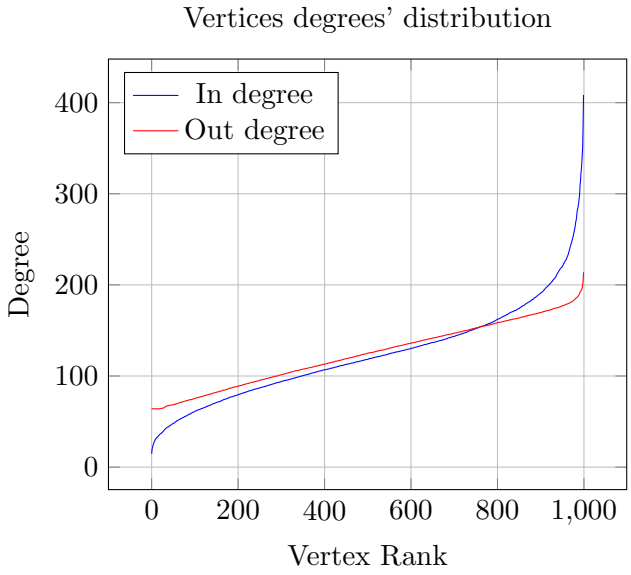

Figure 7: Vertices degrees for 1000 peers, 16 bits addresses, deterministic threshold = 4, keep threshold = 6

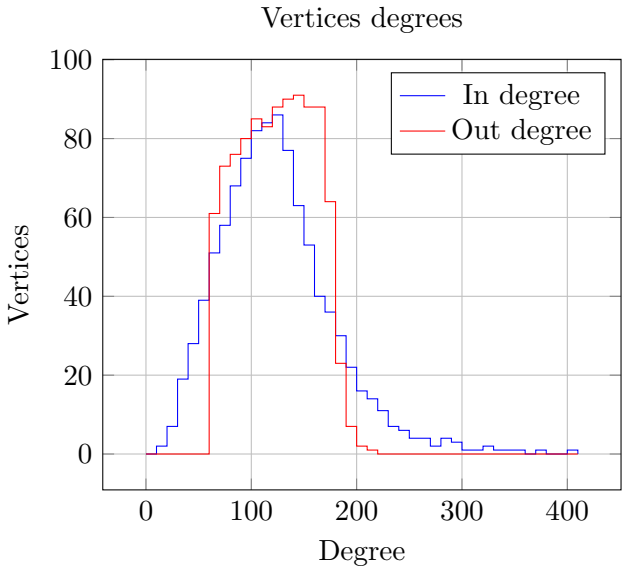

Figure 8: Vertices degrees' distribution for 1000 peers, 16 bits addresses, deterministic threshold = 4, keep threshold = 6

The differences for out degrees come from the differences in the numbers of pulls since last clear. The differences for in degrees come from the distribution of peers on the address plan (address were chosen uniformly at random but the effective distribution is not perfectly uniform).

V.1.b Vertices' weights

Here, for every vertex, we compute separately the sums of the weights of the edges to it and from it. We then sort these numbers in increasing order and plot the result.
For this test, we used a deterministic threshold of 4, a keep threshold of 6 with 16 bits addresses and 1000 peers. The result is the average over 10 runs.

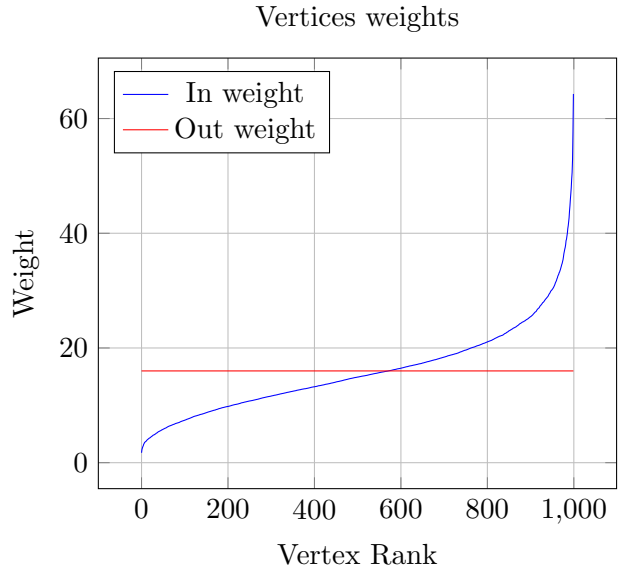

Figure 9: Vertices weights for 1000 peers, 16 bits addresses, deterministic threshold = 4, keep threshold = 6

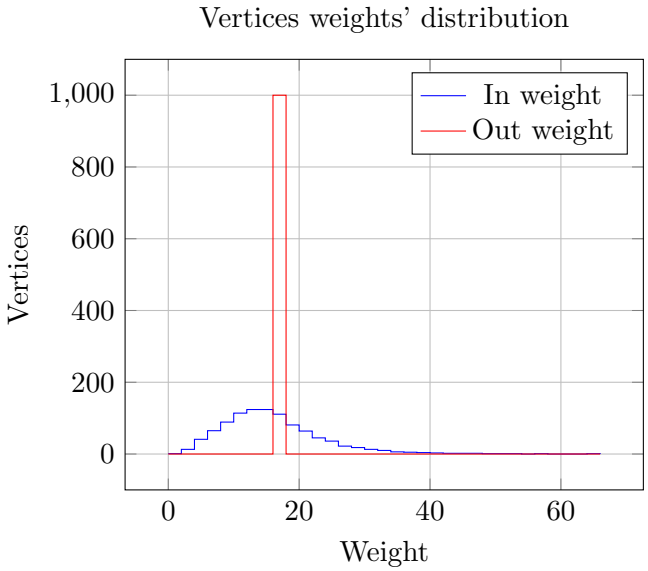

Figure 10: Vertices weights' distribution for 1000 peers, 16 bits addresses, deterministic threshold = 4, keep threshold = 6

The out weights are nearly all the same, due to the lower weights associated to closer peers. The in weights vary greatly since closer peers will have lower weight and isolated (in the address plan) peers are strongly favored.

V.1.c Clustering coefficient

Here, we compute the clustering coefficients of graphs of different size for different values of thresholds for the BPBA-Tree and plot the results.
As metric for the clustering of the of the graph, we will use the *Clemente-Grassi Clustering Coefficient* defined in [CG18]. We choose this generalization of clustering coefficient to directed weighted graph rather than older formulas because of its good properties. Most notably, unlike the one defined in [Fag07], this coefficient will not be deflated for low weight vertices.
Let W be the weight matrix of our graph, A its (unweighted, $\in M(\{0,1\})$) adjacency matrix, s_i^{tot} the sum of the weights of all links to and from the vertex i, d_i^{tot} the total number of links to and from the vertex i and $s_i^{\leftrightarrow} = \sum_{j \neq i} a_{ij} a_{ji} \frac{w_{ij}+wji}{2}$ the sum of the weights (average of in and out weight) of all bilateral links of vertex i. The local Clemente-Grassi Clustering Coefficient of a vertex i, given the matrix W associated with a graph, is defined as follows:

$$C_i^{CG}(W) = \frac{[(W + W^T)(A + A^T)^2]_{ii}}{2[s_i^{tot}(d_i^{tot} - 1) - 2s_i^{\leftrightarrow}]}$$

We use 16 bits addresses and vary other parameters. The result is the average over 10 runs.

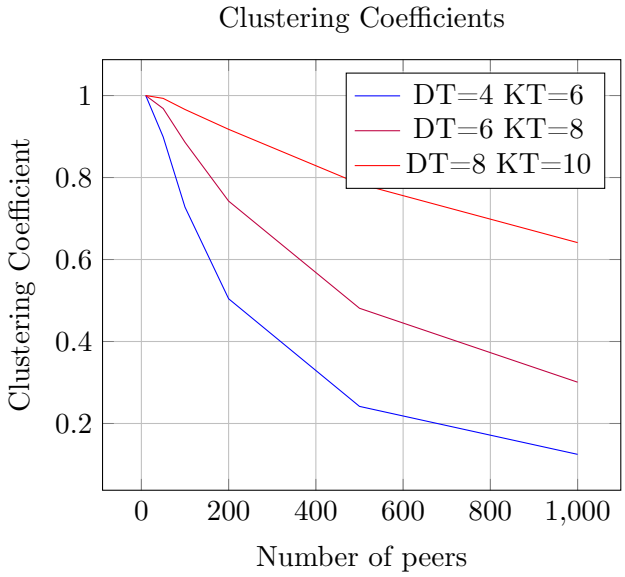

Figure 11: Clustering Coefficients for various parameters values

For a small number of peers, the clustering coefficient is 1, since the graph is complete. This number lowers as the number of peers increases and this effect is stronger with lower thresholds since this reduces the number of links.

V.1.d Average Shortest Path Length

Here, we compute the average shortest path length of graphs of different size for different values of thresholds for the BPBA-Trees and plot the results.
We use 16 bits addresses and vary other parameters. The result is the average over 10 runs.

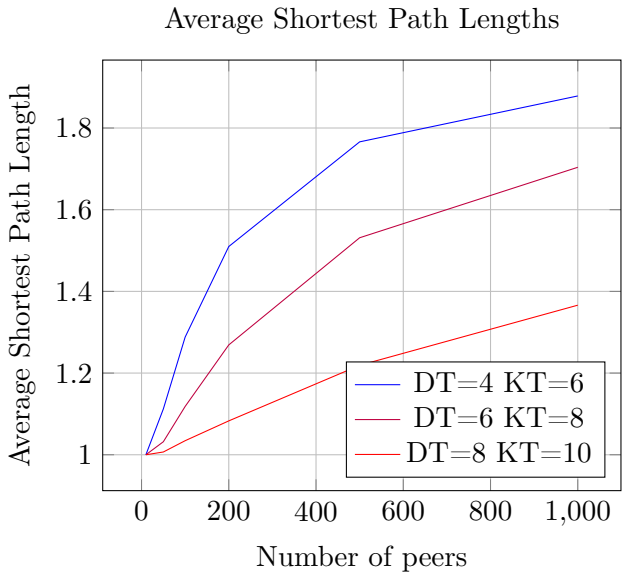

Figure 12: Average Shortest Path Lengths for various parameters values

For a small number of peers, the average shortest path length is 1, since the graph is complete. This number increases with the number of peers and this effect is stronger with lower thresholds since this reduces the number of links.

V.2 Averaging performances

In this subsection, we plot the convergence of the values of peers to the average while performing our averaging algorithm with 3 different peer samplings; HAPS, the perfect (complete graph) peer sampling and a Req-Pull peer-sampling, similar to HAPS except that it is not using a BPBA-Tree (and, thus, do not have the safety properties associated with hierarchical addressing) but a fixed-size queue of peers.
For this test, we used a deterministic threshold of 4, a keep threshold of 6 with 16 bits addresses and 1000 peers. We plotted the values of 40 peers for each case. Peer's values during privacy generation are not shown. For HAPS and Req-Pull, we first let the peer sampling construct local views before starting the averaging process.

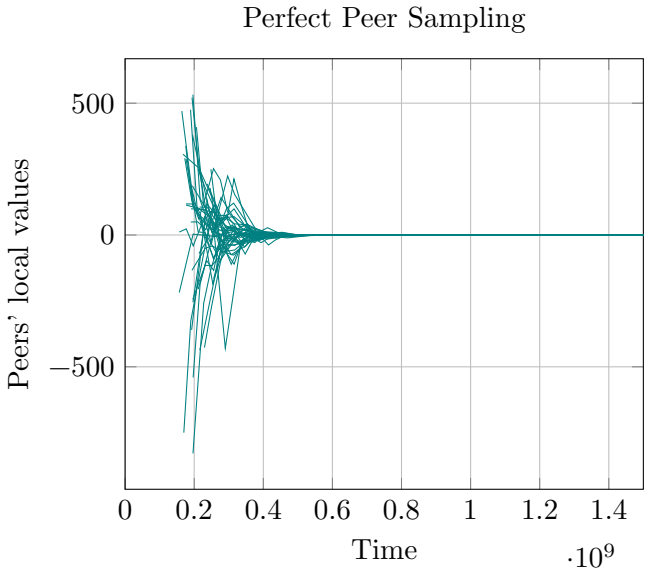

Figure 13: Peer's values convergence with Perfect Peer Sampling

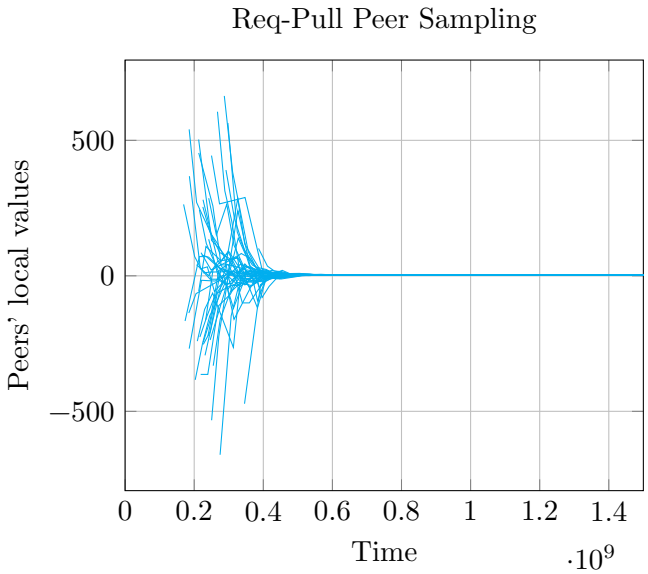

Figure 14: Peer's values convergence with Req-Pull Peer Sampling

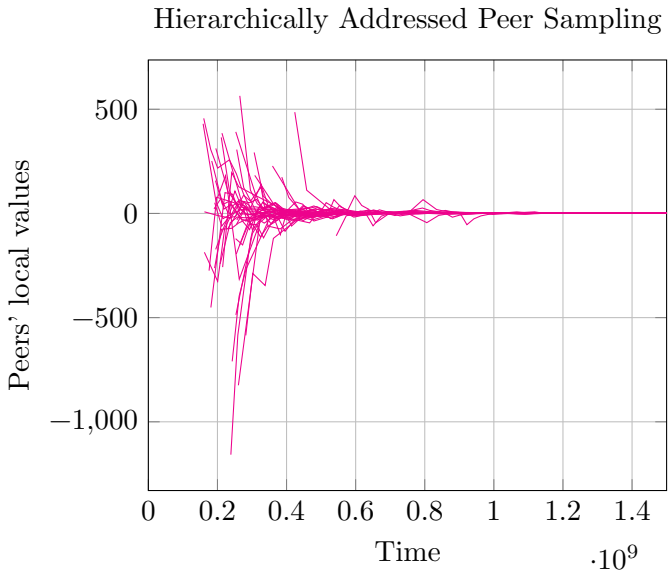

Figure 15: Peer's values convergence with Hierarchically Addressed Peer Sampling

We observe here that convergence is slower with HAPS than with Perfect or Req-Pull peer sampling.

V.3 Global analysis

We see that our new peer sampling protocol will induce uneven distribution of in degree and in weight for randomly distributed addresses, but that is a necessary side-effect of the protection against attacks using close peers.
We observe that, as the number of edges is decreasing (relatively to the total number of possible edges), the clustering coefficient reaches very low values. While obviously increasing with the relative reduction of the number of edges, the average shortest path length remains low.
Finally, even if our peer sampling does not allow the same performances as a more classical Req-Pull peer sampling for averaging, the convergence times remains of similar orders of magnitude and our peer sampling appears to be perfectly usable in practice for this application and, most probably, for others.

VI Conclusion

In this work, we proposed a new variant for a privacy-preserving gossip averaging algorithm. This variant is designed to address the limitations of previous approaches. While such a protocol is necessarily subject to attacks, we formally proved several safety properties of our algorithm.

Since existing peer sampling protocols did not seemed good enough to us for this application, we proposed a new approach of peer sampling: Hierarchically Addressed Peer Sampling. For this approach to work, we defined a new container to replace the classical set used for storing local views. We defined a protocol using this approach and evaluated its performances experimentally.

VI.1 Future works

Now that we have a working averaging protocol, a next step could be to develop a higher level application using the same method for protecting privacy in a gossip environment. Distributed machine-learning seems the main possible application in the near future.

Our approach of peer sampling, using hierarchical addressing, being significantly different of commonly used approaches, it could also be interesting to work on other, possibly more efficient, peer sampling protocols using the same approach.

References

[ADH05] André Allavena, Alan Demers, and John E. Hopcroft. "Correctness of a Gossip Based Membership Protocol". In: *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*. ACM, 2005, pp. 292–301.

[All+16] Tristan Allard et al. "Lightweight Privacy-Preserving Averaging for the Internet of Things". 2016.

[Bor+09] Edward Bortnikov et al. "Byzantine Resilient Random Membership Sampling". 2009.

[CG18] Gian Paolo Clemente and Rosanna Grassi. "Directed clustering in weighted networks: A new perspective". In: *Chaos, Solitons & Fractals* 107 (2018), pp. 26–38.

[Cli+02] Chris Clifton et al. "Tools for Privacy Preserving Distributed Data Mining". In: *ACM SIGKDD Explorations Newsletter* 4 (2 2002), pp. 28–34.

[DBR18] Pierre Dellenbach, Aurélien Bellet, and Jan Ramon. "Hiding in the Crowd: A Massively Distributed Algorithm for Private Averaging with Malicious Adversaries". 2018.

[Dou02] John R. Douceur. "The Sybil Attack". In: *Revised Papers from the First International Workshop on Peer-to-Peer Systems*. Springer-Verlag, 2002, pp. 251–260.

[Fag07] Giorgio Fagiolo. "Clustering in complex directed networks". In: *Physical Review E* 76 (2 2007), p. 026107.

[Fri10] Keith B. Frikken. "Secure Multiparty Computation". In: *Algorithms and Theory of Computation Handbook*. Ed. by Mikhail J. Atallah and Marina Blanton. Chapman & Hall/CRC, 2010, pp. 14.1–14.16.

[HMD12] Zhenqi Huang, Sayan Mitra, and Geir Dullerud. "Differentially Private Iterative Synchronous Consensus". In: *Proceedings of the 2012 ACM Workshop on Privacy in the Electronic Society*. WPES '12. ACM, 2012, pp. 81–90.

[JMB05] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. "Gossip-Based Aggregation in Large Dynamic Networks". In: *ACM Transactions on Computer Systems* 23.3 (2005).

[Lep16] Julien Lepiller. "Private Decentralized Aggregation". 2016.

[MH13] Nicolaos E. Manitara and Christoforos N. Hadjicostis. "Privacy-Preserving Asymptotic Average Consensus". In: *2013 European Control Conference (ECC)*. 2013.

[Néd+17] Brice Nédelec et al. "An Adaptive Peer-Sampling Protocol for Building Networks of Browsers". 2017.

[SKM10] Rashid Sheikh, Beerendra Kumar, and Durgesh Kumar Mishra. "A Distributed k-Secure Sum Protocol for Secure Multi-Party Computations". In: *Journal of Computing* 2 (3 2010), pp. 68–72.