Anonygator: Privacy and Integrity Preserving Data Aggregation

Krishna P. N. Puttaswamy, Ranjita Bhagwan, Venkata N. Padmanabhan

Anonygator: Privacy and Integrity Preserving Data Aggregation

Krishna P. N. Puttaswamy*, Ranjita Bhagwan†, Venkata N. Padmanabhan†

*Computer Science Department, UCSB, †Microsoft Research, India

Abstract. Data aggregation is a key aspect of many distributed applications, such as distributed sensing, performance monitoring, and distributed diagnostics. In such settings, user anonymity is a key concern of the participants. In the absence of an assurance of anonymity, users may be reluctant to contribute data such as their location or configuration settings on their computer.

In this paper, we present the design, analysis, implementation, and evaluation of Anonygator, an anonymity-preserving data aggregation service for large-scale distributed applications. Anonygator uses anonymous routing to provide user anonymity by disassociating messages from the hosts that generated them. It prevents malicious users from uploading disproportionate amounts of spurious data by using a light-weight accounting scheme. Finally, Anonygator maintains overall system scalability by employing a novel distributed tree-based data aggregation procedure that is robust to pollution attacks. All of these components are tuned by a customization tool, with a view to achieve specific anonymity, pollution resistance, and efficiency goals. We have implemented Anonygator as a service and have used it to prototype three applications, one of which we have evaluated on PlanetLab. The other two have been evaluated on a local testbed.

Keywords: distributed aggregation, anonymity, integrity, pollution, tokens, scalability

1 Introduction

Data aggregation is a key aspect of many distributed applications. Examples include aggregation of mobile sensor data for traffic monitoring in a city [20, 23], network performance statistics from home PCs for a network weather service [32], and machine configuration information for a distributed diagnosis system [37].

In such settings, user anonymity is a key concern of the participants. In some cases, this concern is driven by privacy considerations. For example, a user may be willing to have their GPS-enabled phone report traffic speed information from a particular street so long as the system is not in a position to identify and tie them to that location. Likewise, a user may be willing to have their home PC report the performance of a download from `www.badstuff.com` so long as the network weather service they are contributing to is unable to identify and tie them to accesses to possibly disreputable content. In other cases, the

desire for anonymity may be driven by security considerations. For example, a host may reveal local misconfigurations (e.g., improperly set registry keys on a Windows machine) while contributing to a distributed diagnostics system such as PeerPressure [37]. Some of these misconfigurations may have security implications, which would leave the host vulnerable to attacks if its identity were also revealed. Given such security and privacy concerns, an absence of an assurance of anonymity would make users reluctant to participate, thereby impeding the operation of community-based systems mentioned above.

To address this problem, we present Anonygator, an anonymity-preserving data aggregation service for large-scale distributed applications in the Internet setting. The model is that the participating hosts contribute data, which is aggregated at a designated aggregation root node. The data contributed by each node is in the form of a histogram on the metric(s) of interest. For example, a node might construct a histogram of the download speeds it has seen in the past hour over one-minute buckets. All of the histograms are aggregated to construct the probability mass function, or PMF, (which we refer to loosely as the "aggregated histogram") at the server.

Prior aggregation systems such as Astrolabe [33] and SDIMS [38] have focused on achieving scalability and performance by leveraging the participating nodes (i.e., peers) to perform aggregation. While Anonygator also leverages P2P aggregation, it makes several novel contributions arising from a different focus complementary to prior work. First, Anonygator focuses on the issue of providing *anonymity* to the participating nodes while at the same time ensuring that anonymity does not undermine the *data integrity* of the aggregation process. We believe that these are important considerations in the context of distributed aggregation of potentially privacy-sensitive data over nodes that are not all trustworthy. To the best of our knowledge, prior work on P2P aggregation has not considered these issues. Second, Anonygator augments prior work on tree-based aggregation with a novel construct, which we term as a *multi-tree*, that introduces a controlled amount of redundancy to achieve the desired degree of robustness to data pollution attacks. Third, to be flexible in accommodating a range of data aggregation applications, Anonygator includes a *customization tool* to help tune the system to achieve the desired anonymity and data integrity properties while staying within the specified bounds on network communication load.

We present the design of Anonygator, including an analysis of the assurances it provides in terms of anonymity and pollution resistance. We also present experimental results derived from running our implementation on a laboratory testbed as well as on PlanetLab, in the context of a few aggregation-based applications, including resource monitoring, distributed diagnostics and voting.

2 Preliminaries

2.1 Assumptions and Problem Context

We assume a setting where a population of nodes is contributing data, which is then aggregated at a designated aggregation root. The designated root node could be a server that is well-provisioned in terms of bandwidth or could be an

end host that has much more limited bandwidth resources. Even in the former case, the bandwidth demands of aggregation could exceed what the server is able to spare for aggregation. For example, a million nodes, each uploading 1 KB of data every 10 minutes, would impose a bandwidth load of over 13 Mbps on the server for aggregation alone. This means that Anonygator should be able to scale while respecting bandwidth constraints at both the root node and the other participating nodes. In the remainder of this paper, we use the term "(aggregation) server" interchangeably with "(aggregation) root".

We consider the Internet context rather than the sensor network setting that has been the focus of recent work on data aggregation [26, 28, 34]. This means that the typical participating node would belong to a user, who cares about privacy, a consideration largely absent in sensor networks. On the other hand, energy cost, a key consideration in sensor networks, is absent in our context.

We assume that there is an identity infrastructure that grants each participant a public key certificate. This PKI is assumed to exist and operate independently of Anonygator, and grant certificates in a manner that mitigates against Sybil attacks [14] (e.g., by requiring users to provide a credit card number or solve a CAPTCHA when they first obtain a certificate). While Anonygator could choose to use these certified identities as part of the protocol, we assume that the data being aggregated itself does not give away the identity of the source.

We also assume the availability of a trusted entity, which we term as the *bank*, with well-known public key. As we elaborate on in §5.1, the bank issues signed tokens to the participating nodes after verifying their identities. The bank might be the root of the PKI's trust chain or be a separate entity. Regardless, we assume that the bank does *not* collude with the participants in the data aggregation process, including the aggregation root.

While a majority of the participating nodes are honest and cooperate in the operation of Anonygator, up to a fraction, p, of the nodes could be malicious. The malicious nodes, acting individually or in collusion, could try to break anonymity. They could also try to compromise the aggregation process and the final result (i.e., cause "pollution") by injecting large amounts of bogus data themselves or tampering with the data uploaded by other nodes. Note that *we cannot prevent nodes from injecting bogus data* (indeed, determining that the data is bogus may require application-specific knowledge and even then may not be foolproof), *so there would be some pollution even in a centralized aggregation system*, where each node uploads its data directly to the aggregation server, disregarding anonymity. However, the impact of such pollution on the aggregate would be limited unless a relatively large amount of bogus data were injected.

The designated aggregation root, however, is assumed to be honest in terms of performing aggregation; after all, the aggregated result is computed and stored at the root, so a dishonest root node would render the aggregation process meaningless. Nevertheless, the root node, whether it is a server or just an end host, may be curious to learn the identities of the sources, so we need to preserve anonymity with respect to the root as well as the other participating nodes.

The assurance that Anonygator seeks to provide with regard to anonymity and data integrity is probabilistic, under the assumption that the malicious nodes are distributed randomly rather than being specifically picked by the adversary. If the adversary could selectively target and compromise specific nodes, it would not be meaningful to limit the adversary's power to only compromise a fraction p of the nodes. In other words, we would have to assume that such a powerful adversary could target and compromise *all* the nodes, rendering the aggregation process meaningless.

Finally, in the present paper, we do not consider the issue of incentives for user participation in a community-based aggregation system. This is undoubtedly an important issue, but we defer it to future work. Also, given space constraints, we focus our presentation here on the novel aspects of Anonygator's design that have a direct bearing on its security properties. Hence we do not discuss details such as onion route formation [13], random peer selection [24] and decentralized tree construction [38].

2.2 Design Goals

The goals of Anonygator are listed below. Although we state these goals as absolute requirements, we seek to achieve these properties with a high probability.

- **Source Anonymity:** No node in the network, barring the source itself, (i.e., neither the root nor any other participating node) should be able to discover the source of a message.
- **Unlinkability:** Given two messages A and B, no node in the network, barring the source itself, should be able to tell whether they originated from the same source.
- **Pollution Control:** The amount of pollution possible should be close to that in a centralized system.
- **Scalability and Efficiency:** The CPU and bandwidth overhead on the participating nodes and on the aggregation root should be minimized. The system should also respect the bandwidth limits that are explicitly set on the participating nodes, including the root.

2.3 Aggregation via Histograms

As noted in §1, the data to be aggregated is in the form of histograms. For instance, in an application where latency measurements are being aggregated, a host may upload data of the form {50ms: 2, 100ms: 6}, representing 2 samples of 50ms and 6 samples of 100ms.

When performing aggregation, we normalize the individual histograms as probability mass functions (PMFs), before combining the PMFs contributed by all nodes. Normalization ensures that each node receives the same weightage, preventing any one node from unduly skewing the aggregate. So, for example, the histogram in the above example would be normalized to {50ms: 0.25, 100ms: 0.75}. When combined with another normalized histogram, say {75ms: 0.5, 100ms: 0.5}, the aggregate would be {50ms: 0.125, 75ms: 0.25, 100ms: 0.625}. In the rest of the paper, we use the terms PMF and histogram interchangeably.

We believe that the histogram (or PMF) representation of data is quite general and would fit the needs of many applications (e.g., enabling PeerPressure [37] to find the distributions of various registry key settings across a population of hosts). Being an approximation of the probability distribution of a random variable of interest, the aggregated histogram would, for eg., allow us to compute the median value and, in general, the x^{th} percentile, for any value of x.

Histogram-based aggregation does have its limitations. Specifically, it makes it challenging to discover correlations across random variables. For instance, an application may seek to correlate the network failures observed (and reported through Anonygator) by end hosts with the OS being run on the host. Doing so would require computing a histogram with as many buckets as the product of the number of buckets for each variable, leading to a combinatorial explosion in the size of the histogram.

There are also other limitations that arise from our model rather than from our choice of histograms as the basis for aggregation. First, normalizing the histogram would mean we may not be computing the true distribution of a variable. For example, when aggregating download time information for a webpage, a host that downloads the page at a 100 different times (i.e., has 100 samples to offer) would be given the same weight as one that downloads the page just once. However, it is difficult to tell a node that has legitimately performed 100 downloads from one that is merely pretending with a view to polluting the aggregate. Given this difficulty, in Anonygator we choose to normalize, thereby erring on the side of protecting against data pollution, despite the limitation arising from the equal weightage given to all nodes. Second, certain metrics such as the sum, mean, max, and min are not amenable to aggregation in our setting, since a single malicious node can skew the result to an arbitrary extent. Again, this is a problem independent of our choice of histograms as the basis of aggregation.

3 Anonygator Design Overview

Anonygator comprises of three components: (a) *anonymous routing*, to preserve source anonymity, and also ensure unlinkability to a large extent, (b) *light-weight accounting*, to prevent data pollution, and (c) *multi-tree-based aggregation*, to achieve scalability while avoiding the risk of large-scale pollution. The first two components use well-studied techniques, but are essential to the complete anonymity-preserving aggregation system. The third component, the multi-tree, is a novel contribution of our work.

Figure 1 provides an overview of how Anonygator operates. When a source node needs to upload a message (i.e., a histogram) for aggregation, it first obtains tokens from the Bank (1). Then, it attaches a token to the message (2). The token mechanism helps prevent pollution. The source then envelopes the message and the token in multiple layers of encryption to build an *onion* [30], and routes the onion to a tail node, via multiple intermediate nodes (3). The source creates and uses a different onion route for each message to improve the unlinkability of messages. Upon receiving the message, the tail node first validates the token sent with the message, and then passes the message on for aggregation (4). The tail node is part of a distributed structure that we call a *multi-tree*, which performs

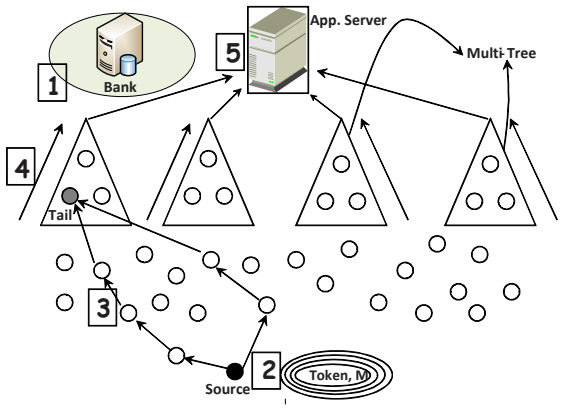

Fig. 1. Design overview of Anonygator.

distributed aggregation on the data. The key idea in a multi-tree, as we will elaborate on later, is to have a many-to-many relationship between parents and children, to help detect any attempts at corrupting the aggregated data. The root of the multi-tree sends the aggregated histograms to the server, which then combines such aggregates from across several multi-trees, if any (5).

Note that the figure shows a *logical* view of our system, for clarity. In reality, any host in the system can be a source, be a tail node for other sources, and also be part of a multi-tree. Also, the tail node that a message is injected into could be in any position in the multi-tree, not just at the leaf. Finally, if we only need anonymity and pollution control, and are willing to sacrifice scalability, the tail node could bypass the multi-tree and upload directly to the server.

4 Anonymous Routing in Anonygator

As described in the overview in §3, a source uses onion routing to convey its message anonymously to a randomly-chosen tail node, which then injects the message into the aggregation tree. To set up an onion path to the chosen tail node, the source uses Tor [13], with the nodes participating in Anonygator serving as the onion routers. §8 discusses how the customization tool chooses the onion route length to achieve specific anonymity and unlinkability goals.

Ideally, we would want to set up a fresh onion path for each message that the source contributes for aggregation. Doing so would minimize the ability of the tail node(s) that receive messages from a source from linking them, even if the same tail node receives multiple messages but over different onion paths. However, setting up an onion path is expensive, since it involves as many public key operations as the length of the onion path. So the overhead of setting up a fresh onion path for each message would be prohibitive.

To resolve this dilemma, we define the notion of an *unlinkability interval*, which is the period during which we wish to avoid reusing any onion paths, making linking messages difficult. However, an onion path can be reused outside of this interval. While such reuse allows tail nodes to link messages, the linked messages would be spaced apart in time, mitigating the impact on unlinkability.

An onion path enables bidirectional communication. Anonygator takes advantage of this to have acknowledgments sent back from the tail node to the source node, which allows the source node to detect events like a message being dropped by an intermediate onion router or a tail node departing the system.

5 Accountability in Anonygator

The drawback of providing anonymity is the loss of accountability. Malicious nodes can "pollute" the data aggregates at the server by uploading large amounts of spurious data without the risk of being black-listed. To prevent this, we introduce accountability in the service via *tokens* and *hash chains*, ideas we borrow from the literature on e-cash, and broadcast authentication [9, 22, 25].

5.1 Anonygator Bank

The *Anonygator Bank* is responsible for maintaining accountability for all data sources in the service. The bank performs two important functions: it supplies the source nodes with a suitable number of token/hash chain combinations and it ensures that the source nodes use these tokens at most once, thus preventing double-spending [9].

Upon joining the network, a node directly contacts the bank and proves its identity in a Sybil-attack resistant manner [6, 14, 15]. The bank then generates a fixed number of signed tokens, based on the node's credentials, and assigns them to the node. When sourcing a message, the node must attach a previously-unused token (or a hash-chain derivative of it, as explained in §5.2) to the message. The limited supply of tokens curtails a node's ability to pollute. The bank also makes sure that source nodes do not double-spend tokens. We explain this procedure in the §5.2 following the explanation of how source nodes use their tokens.

As stated in §2.1 and consistent with previous work [17, 21], we assume that the bank is trusted and that it does not collude with the aggregation server or any other node in the aggregation system. Given this assumption, we believe it is safe for the nodes to divulge their identities to the bank. While the bank knows the identity of the sources, their capabilities, and their token usage, it does not know anything about the data and messages that the sources generate. The aggregation server, on the other hand, has access to the data, but it does not know the identity of the sources. This helps us achieve our anonymity goals.

5.2 Using Tokens and Hash Chains

A source node with a data item to be aggregated includes a token, signed by the bank, along with the data item to generate a message, M. It routes this message to tail node T, as explained in §4. T first verifies that the bank has indeed signed the token (one asymmetric cryptography operation) and then contacts the bank to ensure that the token has not already been used. The bank performs this check by treating the token (or its ID) as an opaque blob of bits that is looked up in a local data structure. If the bank informs T that the token was not previously used, T deems the corresponding data item as valid and forwards it on for aggregation. Otherwise, T discards the data item.

Although the verification mechanism described above does provide anonymous accountability, it involves an asymmetric operation and communication

with the bank per message, which can be quite expensive, especially if the message generation rate is high. Anonygator uses *hash chains* along with tokens to reduce this overhead. A hash chain [22] is a chain of hash values obtained by recursively hashing a random *seed* using a unidirectional hash function like SHA1. The final hash after y hashes is called the *head* of the hash chain. The contents of a token augmented with hash chain information are: $Token_i = \{ID_i, head_i, sign(hash(ID_i \ . \ head_i))\}$ where ID_i is the token ID, $head_i$ is the head of the hash chain generated for this token, and both token ID and the head are signed by the bank.

With this token construction, the modified algorithm to upload messages by a source S via a tail T is as follows: The first time source S sends a data item to tail T, it includes a token with id ID_T with the data. T performs the verification of the token as mentioned earlier. In all subsequent messages that S sends to T, S includes only the tuple (ID_T, H_x) in decreasing order of x. T just needs to verify that for token ID_T, it receives a message with the hash value H_x *only after* it has received a message with hash value H_{x+1}. It can do so simply by applying the hash function to H_x and verifying that the value matches H_{x+1}.

As a result, of all messages that S sends to T, only the first message involves an asymmetric cryptographic operation and direct communication with the bank. Note that, as explained in §4, the source uses the same onion path to communicate with T, thereby allowing messages to be linked. So there is *no* additional diminution of unlinkability because of using the same token.

5.3 Token Management under Churn

Once a source uses a token with a certain tail node, Anonygator does not allow the source to use that token with another tail node. The token is, therefore, "tied" to the tail node on which the source first used it. So if this tail node leaves the system, say due to churn, the source cannot use even the unused portion of the hash chain associated with this token, with any other tail node.

To avoid this problem, we introduce the notion of an *epoch* (T_e). Each source node obtains a set of tokens from the bank at the start of an epoch. At the end of each epoch, all tail nodes report to the bank the last hash value they received (i.e., the last value that was expended by another node) for every token ID. So the bank knows the extent to which each token was used. For example, if the length of a token's hash chain is 100 and the last value reported by a tail node is the 30th one in the chain (counting from the head), the bank can deduce 30 values have been used and 70 values remain unused.

After two epochs, the bank tallies how many hash chain values have been used for each token, and provides a "refund" to source hosts for the remainder. A refund is nothing but appropriate accounting at the bank to reflect the partial- or non-use of a token, so that the source can get a fresh token issued to it while remaining within any quotas imposed by the bank. Since accounting at the bank depends on the tail nodes reporting usage of tokens, there is the risk of a *spurious refund attack*, where a tail node, in collusion with the source node, fails to report the usage of a token. To address this issue, Anonygator introduces redundancy, including at the level of tail nodes, as we elaborate on next.

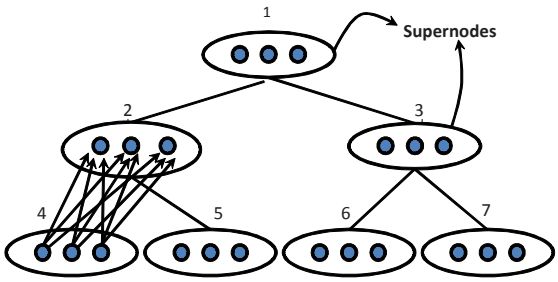

Fig. 2. The structure of a multi-tree.

6 Distributed Aggregation Using Multi-Trees

In this section, we address the issue of scalability of data aggregation in Anonygator. Using tree-based aggregation is a natural way to improve scalability of aggregation: as data flows from the leaves to the root, the data gets aggregated, and the root receives aggregated data while processing incoming traffic only from a small set of nodes. However, using a regular trees for aggregation raises several security concerns. For example, a *single* malicious node near the root in the tree can completely change the aggregate histogram from the entire sub-tree below it. This can cause unbounded amount of pollution.

In order to be robust against such attacks, we propose a distributed aggregation mechanism using a structure that we call a *multi-tree*, as in Figure 2. The idea in a multi-tree is to group together the nodes into *supernodes*, each containing a mutually-exclusive set of k nodes. These supernodes are organized as a regular tree. A parent-child relationship between two supernodes is translated into a parent-child relationship between every member of the parent supernode and every member of the child supernode.

The system supports a set of such multi-trees, as shown in Figure 1. The exact number of multi-trees depends on the bandwidth of the aggregation server, as we analyze in §8.2. The node membership of each multi-tree is non-overlapping with respect to the membership of the other multi-trees.

6.1 Data Injection

Every node that serves as a tail node for the purposes of anonymous routing (§4) is a member of a supernode in the tree. Even though new data to be aggregated (i.e., a histogram) is introduced into the multi-tree at a tail node, the supernode that the tail node is a member of can be at any level of the multi-tree, not necessarily at the leaf level.

The source node sends the tail node the new data to be aggregated, along with k tokens, one for the tail node itself and one each for the $k-1$ other nodes in the tail node's supernode. The tail node then forwards the histogram along with one token to each of the other $k-1$ nodes in its supernode. If the membership of the super node changes (say because of node churn), the tree node informs the source through the onion route, so that the source can send fresh tokens, rather than just new hash values, for each new node in the supernode.

The above procedure mitigates against the spurious refund attack noted in §5.3, as we discuss in detail in §7.3. Also, an alternative to routing its message

via a single tail node would be for the source to send separate messages, along with their respective tokens, directly to each node in the tail node's supernode. This *k-redundant algorithm* increases messaging cost by a factor of k but reduces the risk of pollution, as we discuss in detail in §7.2.

6.2 Data Aggregation

The objective of having k nodes within each supernode is to be able to compute the correct aggregate histogram with high probability, even in the presence of malicious nodes. Figure 2 shows a sample multi-tree with $k = 3$. In this example, each host in supernode 4 (on the bottom left) uploads its histogram to each host in supernode 2. Each node in supernode 2 therefore receives $k = 3$ histograms from supernode 4. If all nodes in supernode 4 were honest, the $k = 3$ histograms received by each node in supernode 2 would be identical. However, in the presence of malicious nodes, these histograms would diverge, as we discuss next.

At the end of a time period that we call an *aggregation interval*, each node in supernode 2 picks the histogram that is repeated at least $\lfloor \frac{k}{2} \rfloor + 1$ times (2 times, in this example). Histograms that do not meet this minimum count are discarded. Therefore, for a supernode to accept a bogus histogram, more than half the nodes in its child supernode would have to be malicious and colluding. Every parent supernode determines such "majority" histograms for each of its child supernodes and then combines these to compute an aggregate histogram representing data received from all of its child supernodes. For example, supernode 2 in Figure 2 combines the majority histograms from supernodes 4 and 5 to compute an aggregate histogram. Each node in supernode 2 then uploads this aggregate histogram to all k members in its parent supernode (supernode 1, here), and the process repeats. Having the parent do the voting is necessary. Putting the onus of voting on the parent avoids the complexity and obviates the need for distributed voting.

If each supernode in the multi-tree has a majority of non-malicious nodes, then the multi-tree is said to be "correct", since the correct overall aggregate histogram is produced, despite the presence of malicious nodes. Given the probability of aggregate correctness P_c, which is the probability that the aggregate the multi-trees produce are correct, Anonygator's customization tool determines the best multi-tree configuration that will satisfy this requirement (§8.2).

7 Attacks and Defenses

In this section we discuss several attacks on Anonygator, their impact, and potential defenses against them. We consider the possibility of attacks by source nodes, relay nodes (i.e., onion routers), tail nodes, and nodes in the aggregation tree. These attacks could be aimed at compromising either security (in terms of anonymity or unlinkability) or data integrity (in terms of pollution control). Our focus here is on attacks that are specific to the various mechanisms in Anonygator. For attacks on the underlying Tor system, we refer the reader to [13].

7.1 Attacks by Malicious Source Nodes

Direct Data Injection Attack [8] This occurs when a source node directly injects erroneous data in the legitimate messages it generates. As explained in §2.1, the

server cannot, in general, tell that the data is erroneous. However, the token mechanism in Anonygator (§5.2) limits the amount of data that a node can contribute for aggregation. Hence the *pollution bound* for this attack, i.e., the fraction of data injected that could be spurious, is the same as the fraction, p, of the nodes that are malicious.

Interpreting this pollution bound for histograms, we can say that the histogram will be at most p percentile off from the ground truth. For example, if $p = 0.01 = 1\%$, then the median value in the aggregate histogram would give us a value that lies somewhere in the range of the 49^{th} to the 51^{st} percentiles in the true, pollution-free histogram.

Fake Token Attack A malicious source could send a flood of messages, each tagged with a fake token, to one or more tail nodes. The tokens and the associated messages are eventually rejected, so data pollution does not occur. However, the attacker intends to tie down computational and network resources at the tail nodes in checking these fake tokens, i.e., attempt resource exhaustion. Existing techniques such as client puzzles [12] could be used by tail nodes as defense when the rate of message receipt is very high.

7.2 Attacks by Malicious Tail Nodes

Message Replacement Attack A malicious tail node can take the data that a source node sends them and replace it with spurious data. Since a source picks a malicious tail node with probability p, the fraction of data items potentially affected by message replacement attacks is p. Since this is in addition to the pollution of p possible with the direct data injection attack discussed above, the total pollution bound is $2p$.

However, the source could send copies of its message independently to each node in the tail node's supernode (the k-redundant algorithm from §6.1), thereby denying the tail node the ability to subvert aggregation by doing message replacement. This would mean that the overall pollution bound would remain p (rather than $2p$), but this would come at the cost of increased messaging cost.

Finally, note that the message replacement attack subsumes other attacks where a malicious tail node drops the received messages, forwards the tokens in these messages to a colluder, who later uses the tokens to cause pollution.

Spurious Churn Attack As noted in §6.1, whenever there is churn in the tail node's supernode, the source has to obtain and send fresh tokens, one for each "new" node in the supernode. A malicious tail node can try to exhaust the source node's quota of tokens by pretending that there is churn when there is none. To defend against such an attack, a source node can determine that a particular tail node is reporting much higher churn than is the norm and hence decide to switch to using a different tail node.

7.3 Attack by Colluding Source and Tail

Spurious Refund Attack The attack involves a source node contributing data for aggregation but, in collusion with a tail node, avoiding expenditure of tokens (or, equivalently, obtaining a refund of the tokens spent, as noted in §5.3). However, as noted in §6.1, Anonygator requires a majority of (honest) nodes in the chosen

tail node's supernode to receive and forward a source's data up the tree, for it to be included in the aggregation process. Hence the source will have to expend at least $\lfloor \frac{k}{2} \rfloor + 1$ tokens, even if not the full complement of k tokens, which means less than a 2x savings in terms of token expenditure. Also, note that by expending just $\lfloor \frac{k}{2} \rfloor + 1$ tokens, the source would run the risk of having its data be discarded in the aggregation process if any of the $\lfloor \frac{k}{2} \rfloor + 1$ nodes that it sent the token to turns out to be dishonest.

7.4 Attack by Malicious Relay Nodes

Message Dropping Attack A relay node, i.e., an onion router, could drop a message that it is supposed to forward on a path leading to a tail node. However, as noted in §4, the bidirectionality of onion paths allows the source node to detect such drops by looking for an acknowledgment from the tail node. Even if such an acknowledgment mechanism were not in place, the worst that the malicious relay node could so is to drop messages randomly, without knowledge of either the source or the contents. Such dropping would, therefore, be no worse than random network packet drops.

7.5 Attack by Malicious Tree Nodes

A malicious tree node could attempt a data injection attack or a message replacement attack with a view to subverting the aggregation result. However, such an attack would not be successful unless a majority of nodes in supernode were malicious and colluding. As we explain in §8.2, Anonygator's customization tool ensures that the likelihood of such an occurrence is below the bounds specified by the application designer.

8 Configurability in Anonygator

This section describes how an application designer can configure Anonygator to best suit the application's needs of anonymity, unlinkability and correctness. Anonygator's customization tool (CT) can configure anonymous routing, token usage and multi-trees to meet the application's requirements while not exceeding the amount of computing and network resources that participating hosts in the system are willing to contribute. We describe how the CT works in this section.

8.1 Customization Tool Overview

Figure 3 shows part of the functionality of the customization tool of Anonygator. The first-class properties that an application needs to specify to Anonygator are *anonymity* (A), *unlinkability* (P_U) and the *probability of correctness* (P_C). The metric for probability of correctness, as mentioned in §6, is the probability that the distributed aggregation generates the correct aggregate histogram.

Apart from these, the application has other properties that the designer inputs to the tool. The *histogram generation rate* (R_{hg}), specified in histograms per second, provides the average rate at which sources generate messages or histograms. The *histogram aggregation rate* (R_{ha}), also specified in histograms per second, is the rate at which hosts upload aggregate histograms to their parent supernodes in the multi-tree. For simplicity, we assume that all histograms are of the same size, though in reality, there would be variations based on applications.

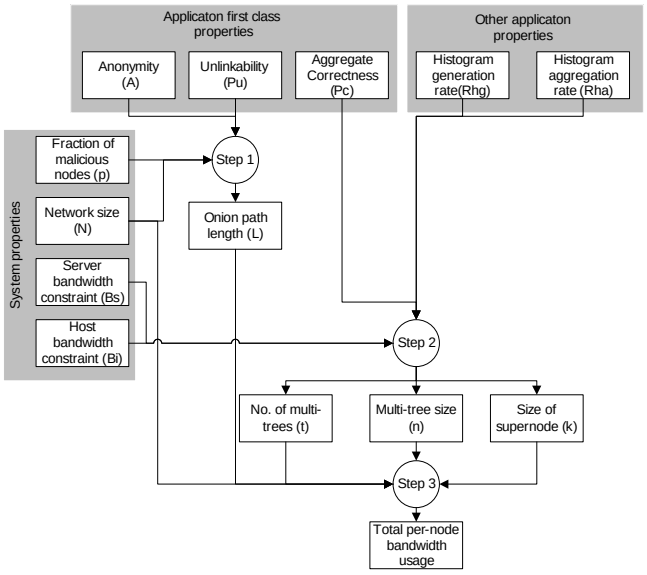

Fig. 3. Procedure used by the CT to determine the total per-node bandwidth overhead.

The *unlinkability interval* (T_l) is the time interval after which a source node can reuse a previously used onion path or use the next value of an already-used hash-chain. The *epoch length* (T_e) is the duration of time for which tokens are valid and it dictates the periodicity with which the bank assigns fresh tokens.

The designer also inputs several host characterization parameters that define the system's properties. As shown on the left of Figure 3, these parameters include the fraction of malicious nodes (p), the size of the Anonygator network or the number of nodes participating in the network (N), the maximum incoming server bandwidth dedicated to aggregation (B_s) specified in histograms per second, and the maximum incoming host bandwidth (B_h) also specified in histograms per second. Given the application's requirements and system specification, the CT informs the designer of the anonymous route length and multi-tree structure through Steps 1 and 2 in Figure 3.

The churn rate (R_c) is a measure of the number of hosts that leave the system per second (as estimated by the system designer). The key setup rate (R_{ks}) is the rate at which a source node can perform asymmetric cryptographic operations to perform onion path setup discussed in §4. Step 3 of the CT calculates the number of tokens required per epoch and hash chain length per token using churn rate, key setup rate, unlinkability interval, histogram generation rate and epoch length. In this section however, we concentrate on the details of just Step 2. We refer the reader to [29] for the details of the analysis of Step 1, and Step 3. Since this analysis is similar to the analysis in prior work on anonymous communication, and due to space constraints, we leave it out of this paper. Finally, Table 1 has a summary of the symbols we use in the following subsections.

Symbol	Definition	Type
A	Anonymity (Entropy Ratio)	input
P_U	Unlinkability (Probability)	input
P_c	Aggregation correctness probability from multi-tree	input
N	Total number of hosts	input
p	Fraction of malicious nodes	input
B_s	Maximum incoming server bandwidth (Histograms/Sec)	input
B_h	Maximum incoming host bandwidth (Histograms/Sec)	input
f	Fanout of the multi-tree	input
R_c	Churn rate in system (Nodes/Second)	input
R_{hg}	Histogram generation rate (Histograms/Sec)	input
R_{ha}	Histogram aggregation rate (Histograms/Sec)	input
T_l	Unlinkability interval (Seconds)	input
T_e	The Epoch length (Seconds)	input
R_{ks}	Max. rate of key setup for a node (Numbers/Sec)	input
L	Length of the onion path	output
k	No. of nodes in a supernode	output
n	No. of supernodes per multi-tree	output
t	No. of multi-trees	output
N_t	No. of tokens	output
L_H	Hash chain size	output

Table 1. Variables used by the customization tool.

8.2 Step 2: Probability of Correctness to Multi-tree Structure

In this section, we summarize how the CT calculates multi-tree structure. The CT calculates a feasible region for the the number of supernodes in a multi-tree (n) based on three constraints. First, the number of supernodes has to be small enough to satisfy the probability of correctness: the more the number of supernodes, the higher the probability of a "bad" supernode with more than $\lfloor \frac{k}{2} \rfloor + 1$ malicious nodes. Second, the number of supernodes is limited by the number of hosts participating in the system. Third, the number of supernodes needs to be large enough such that nodes in a supernode use less than their maximum specified incoming bandwidth (B_h). These three bounds are expressed in Eq. 1, Eq. 2, and Eq. 3 respectively.

$$n \leq \frac{logP_c}{t.logP_{sn}}, P_{sn} = \sum_{i=\lfloor \frac{k}{2} \rfloor + 1}^{k} \binom{k}{i} (1-p)^i p^{k-i} \tag{1}$$

$$n \leq \frac{N}{kt} \tag{2}$$

$$n \geq \frac{NR_{hg}}{t(B_h - R_{hg}L + fkR_{ha})} \tag{3}$$

The CT determines a relation between k and t given the incoming bandwidth capacity B_s set apart by the server for aggregation. The CT calculates the maximum multi-trees that can directly upload data to the server as

$$t = \left\lfloor \frac{B_s}{k.R_{ha}} \right\rfloor \tag{4}$$

This is because each multi-tree's root supernode (with k nodes within) uploads $k.R_{ha}$ aggregate histograms to the server per second. Therefore, the server can dedicate at most $\frac{B_s}{k.R_{ha}}$ bandwidth to each multi-tree.

Figure 4 shows the feasible region for n for different values of k plotted using Equations 1, 2 and 3. The input parameters set are: $p = 0.01$, $N = 1$ million, $P_c = 0.90$, $B_s = 100{,}000$ histograms/sec, R_{ha} is 10 histograms/sec, R_{hg} is 100 histograms/sec, and $B_h = 5000$ histograms/sec. Note that f is fixed at 3. Based on these constraints, the CT chooses the minimum value of k that makes n fall in the feasible region. In this example, this value is around 16, translating to a value of 45 for n, and 625 for t.

It is possible, though, that for some input values, there are no feasible values of n, k, and t. In such cases, the CT alerts the application designer that their system requirements are too high to be met, and that they need to revise their application or system properties.

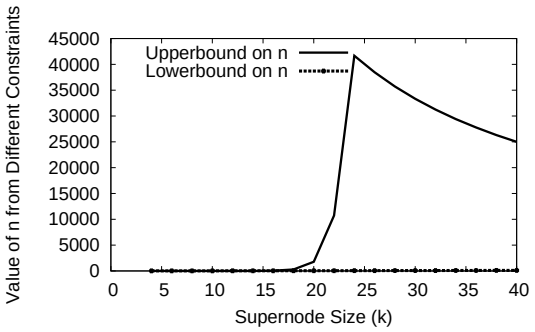

Fig. 4. Visual depiction of the multi-tree constraints.

9 Implementation

Implementation Status: We have implemented Anonygator on two platforms: our first implementation, built for a Linux testbed, consists of roughly 1400 lines of Python code and uses the pycrypto library for the base cryptographic functions. Our second implementation, built on the .Net framework, consists of 2400 lines of C# code and uses the BouncyCastle [5] cryptographic library. We use RSA (1024 bits keys) as the asymmetric cipher and Rijndael as the symmetric cipher.

The Anonygator implementations provide a library that supports all three components of Anonygator: anonymous routing, data pollution prevention, and multi-tree based aggregation. Currently, in our prototypes, all node discovery and multi-tree construction operation is centralized: a directory service informs nodes of hosts that they could use as tail nodes. The directory service also determines the membership of the multi-tree by assigning hosts to the different supernodes in the multi-tree. However, both node discovery and multi-tree construction

could be decentralized using techniques such as DHTs [31] and distributed tree construction algorithms [7, 38].

Anonygator API: Table 2 lists the APIs that Anonygator provides: the first two API calls are for the client side, and the last two for the server side. Note that the Anonygator API enables an application's client only to *send* messages anonymously, and the aggregation server to *receive* and *aggregate* these messages. Separately, the application designer uses the customization tool to tune Anonygator's parameters (§8).

API	Purpose
initAnonygatorClient() (Client)	Buys tokens, installs keys.
sendData() (Client)	Sends app. data to tree.
initAnonygatorServer() (Server)	Inits agg. server
pushdownConfiguration() (Server)	Sends multi-tree configuration to the clients.

Table 2. APIs of the Anonygator library. Client and Server side calls are marked.

10 Evaluation

To evaluate Anonygator, we have implemented three applications on two separate testbeds. The first application, inspired by systems that measure resource usage on distributed hosts [1], aggregates CPU usage on various hosts over time. The second application, inspired by FTN [19, 37], involves aggregating machine configuration parameter settings across a population of hosts. The third is a voting application motivated by Credence [36], a distributed object reputation system. We implemented and evaluated the CPU Aggregation on PlanetLab, while the other two are evaluated on a Windows Vista cluster testbed.

10.1 Aggregation of CPU Utilization

Using our Linux implementation, we have built an application to aggregate percent CPU utilization on a distributed set of Planetlab hosts. The purpose is to understand how the distribution of percent CPU utilization on PlanetLab varies over time. The histograms that the application generates and aggregates consist of buckets at 10% increments, i.e. the first bar represents the fraction of hosts with 0-10% CPU usage, the second bar represents the fraction with 10-20%, etc.

Algorithm 1 shows the pseudo-code for the client side of this application. After initialization, the client periodically uploads its CPU utilization using the `sendData` call. Our purpose in presenting the algorithm is to show that the code required for implementing this application atop the Anonygator API is fairly simple since the anonymity preservation, and token accounting is done entirely by the Anonygator library.

Since the application itself provides similar functionality as other monitoring systems such as CoMon [1], we refrain from delving into the actual measurements that the application gathers. Instead, we concentrate on evaluating the scalability and bandwidth usage of the components of the Anonygator system itself.

Bank Scalability: Our first experiment evaluated the scalability of the Anonygator bank – the rate at which the bank generates tokens. We found this to be the most resource-intensive function for the bank since each token generation

Algorithm 1 The Algorithm for Aggregation of CPU Utilization on the Client.

Contribute_Data()

```
1: initAnonygatorClient() /* Initialize Anonygator */
2:
3: while 1 do
4:    readCPUUsage() /* Read CPU Usage */
5:    sendData(data) /* Send out data via Anonygator */
6:    sleep(uploadInterval)
7: end while
```

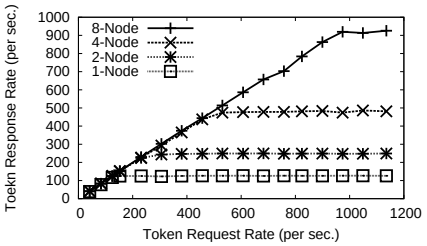

Fig. 5. Bank's scalability in terms of token generation rate.

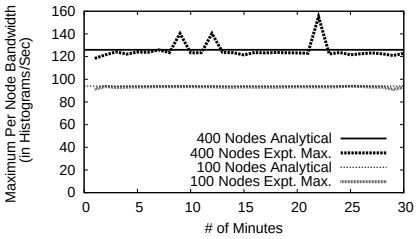

Fig. 6. Maximum incoming bandwidth usage over all hosts in the interior supernodes of the multi-tree.

involves one asymmetric crypto operation and generating the head of the hash chain by performing L_H hashes (token verification is more than 20 times faster than generation). In our experiment, we set L_H, the hash chain length, to 1000.

The Anonygator bank was running on a cluster of machines at the University of California, Santa Barbara. Each machine has a 2.3 GHz Intel Xeon processor and 2GB memory. All machines ran 32-bit RedHat CentOS and were interconnected through a Gigabit Ethernet switch. We varied the number of machines that constituted the Anonygator bank between 1 and 8. The clients ran on 100 PlanetLab nodes, which periodically contacted and received tokens from the bank. We varied the rate at which the clients requested tokens from the bank.

Figure 5 shows that the rate at which the bank generates tokens varies linearly with the number of machines in the cluster. With 1 machine, the peak rate is 125 tokens/sec, with 2, it is 248 tokens/sec, with 4 it is 486 tokens/sec and with 8, it is 919 tokens/sec. These results follow from the fact that creating a hash-chain of size 1000 takes 3.75 ms and signing the token takes 4.25 ms, for a total of 8ms to generate one token. This implies that with 1,000,000 hosts in the system, the 8-machine bank can support a histogram generation rate of 0.919 histograms per second, or 55 histograms per minute. For many aggregation systems [2, 4], this is a fairly high rate of data generation. The capacity of the bank can be further improved by increasing the hash chain length or the cluster size.

Host Bandwidth Usage: Next, we evaluated whether the maximum incoming host bandwidth on each PlanetLab machine was indeed capped by the value

Total number of nodes	L	k	n	t
100 nodes	3	4	25	1
400 nodes	3	6	66	1

Table 3. Outputs from the CT for PlanetLab deployment.

input to the customization tool. We created two deployments of the application on PlanetLab, one with 100 hosts and the other with 400 hosts, and ran the application on each of these two host sets for 30 minutes. The value of B_h was set to 94 for the 100 node deployment and 126 for the 400 node deployment. We set the required anonymity A to 0.99, the unlinkability P_U to 0.99, and probability of correctness P_c to 0.7. The number of hosts, N, was set to 100 or 400 depending on the experiment, and the fraction of malicious hosts, p, was 0.05. We set the histogram generation rate to 10 per minute, and the histogram aggregation rate to 2 per minute. The histogram size is approximately 40 bytes, since each histogram has 10 bars and the size of each bar is 4 bytes (int).

In the onion routing phase, the message size due to the onion encapsulation is roughly 400 bytes. While this may seem high, we believe that the overhead is manageable since the histogram generation rate from source to tail node is set to only 10 per minute. At extremely high rates of histogram generation, however, this overhead could significantly affect performance. However in our experience, aggregation-based systems [2, 4, 11] do not have extremely high data generation rates *per-source* (though the bandwidth usage at the server, with a large number of sources could be significant). The maximum incoming server bandwidth, B_s, was set to 8 histograms per minute for the 100 node experiment and to 12 histograms per minute for the 400 node case. Table 3 shows the value of the various output parameters the CT calculated with these inputs.

Some of these parameters (such as low values of server bandwidth, low correctness probability, and low histogram generation and aggregation rates) are not representative of what one may expect in a real deployment. However, since the experiment's objective was to evaluate the bandwidth usage on a host, we needed to set parameters that created multi-level trees with just hundreds of hosts at our disposal. With our choice of parameters, the 100-node deployment had a multi-tree with 4 levels and the 400-node deployment had 5 levels.

Figure 6 shows a time-series of the *maximum* instantaneous bandwidth (calculated over 1 minute buckets) on a node, calculated over all nodes in the system. Hence each data point comes from the node whose bandwidth usage is maximum in that minute. The figure shows that our implementation of the Anonygator system does conform to the bandwidth constraint specified in both experiments thereby confirming the effectiveness of the customization tool. The three spikes correspond to short-term variability in bandwidth usage on certain nodes: the nodes with maximum bandwidth usage were significantly under-utilized in the minute just prior to the spike.

We performed a similar study to evaluate usage of server bandwidth B_s which yielded similar results. We leave out the experiment details due to lack of space.

10.2 Distributed Diagnostics and Voting Application

We implemented two more applications – a distributed diagnostic application, inspired by FTN [19] and a voting application inspired by Credence [36]. We also deployed them on a lab cluster of 25 machines. We leave out the details due to space limitations, but refer the reader to our technical report at [29].

11 Related Work

Several mobile data collection systems such as CarTel [20], Mobiscopes [3], and Nericell [23] involve sensors uploading data periodically to a central repository. Protecting the privacy of the contributors would be important but has not received much attention in most of these systems.

A notable exception is AnonySense [11], which provides privacy while assigning tasks to mobile sensor nodes and retrieving reports from them. However, AnonySense does not perform data aggregation. SmartSiren [10] collects reports at a server and uses them to detect viruses using certain thresholds. It attempts to provide anonymity to clients, and also describes the problem of pollution control during anonymous report submissions. SmartSiren however assumes that submitting reports via the IP network provides sufficient anonymity. Also, it uses random ticket exchange between clients to avoid the server from tracking smartphones based on tickets.

Several recent systems have used tokens to achieve accountability [16, 27, 35]. The tokens used in these systems are always involved in the critical paths. Thus, the clients need to contact the bank (and verify the token) for every application-level operation (exchange a block [27], accept a mail [16, 35], etc.). However, Anonygator clients need to contact the bank once to verify a token and all subsequent messages are authenticated offline using hash chains, making the bank in Anonygator much more scalable.

A recent work [39] proposed mechanisms to provide anonymity and accountability in P2P systems. However the computational and bandwidth cost of this approach is significantly higher than Anonygator due to its reliance on heavyweight cryptographic constructs.

Several systems explore the problem of performing secure data aggregation in sensor networks [8, 18, 26, 28]. But the mechanisms used in sensor network do not provide anonymity to the contributing sensor nodes. The base station either receives the data from the nodes directly, or shares a unique key with the nodes and hence can easily link the data to nodes.

12 Conclusion

In this paper, we have presented Anonygator, a system for anonymous data aggregation. Anonygator uses anonymous routing, token and hash-chain based pollution control, and a multi-tree based distributed aggregation scheme, to build a scalable, anonymous aggregation system. Anonygator's customization tool allows the designer to meet the desired anonymity and unlinkability goals, while honoring the specified pollution bounds and bandwidth limits. We have built three applications on Anonygator and have tested them on PlanetLab and a local cluster of machines.

References

1. CoMon webpage. http://comon.cs.princeton.edu.
2. Microsoft Online Crash Analysis. http://oca.microsoft.com/en/dcp20.asp.
3. Abdelzaher, T., et al. Mobiscopes for human spaces. *IEEE pervasive computing* (2007).
4. Aggarwal, B., Bhagwan, R., Das, T., Eswaran, S., Padmanabhan, V., and Voelker, G. NetPrints: Diagnosing Home Network Misconfigurations using Shared Knowledge. In *Proc. of NSDI* (2009).
5. Bouncycastle. The legion of the bouncy castle. http://www.bouncycastle.org.
6. Castro, M., et al. Security for structured peer-to-peer overlay networks. In *OSDI* (December 2002).
7. Castro, M., et al. Splitstream: high-bandwidth multicast in cooperative environments. In *Proc. of SOSP* (2003).
8. Chan, H., Perrig, A., and Song, D. Secure hierarchical in-network aggregation in sensor networks. In *Proc. of CCS* (2006).
9. Chaum, D. Blind signatures for untraceable payments. In *Proceedings of Crypto* (1982), vol. 82, pp. 23–25.
10. Cheng, J., Wong, S., Yang, H., and Lu, S. Smartsiren: Virus detection and alert for smartphones. In *MobiSys* (2007).
11. Cornelius, C., et al. AnonySense: Privacy-aware people-centric sensing. In *MobiSys* (2008).
12. Dean, D., and Stubblefield, A. Using client puzzles to protect TLS. In *Proceedings of the 10th USENIX Security Symposium* (2001).
13. Dingledine, R., Mathewson, N., and Syverson, P. Tor: The second-generation onion router. In *USENIX Security Symposium* (Aug 2004).
14. Douceur, J. R. The Sybil attack. In *IPTPS* (2002).
15. Dwork, C., and Naor, M. Pricing via processing or combatting junk mail. *LNCS* (1993).
16. Gummadi, R., et al. Not-a-bot (nab): Improving service availability in the face of botnet attacks. In *Proc. of NSDI* (2009).
17. Hoh, B., et al. Virtual trip lines for distributed privacy-preserving traffic monitoring. In *MobiSys* (2008).
18. Hu, L., and Evans, D. Secure aggregation for wireless networks. In *Workshop on Security and Assurance in Ad hoc Networks* (2003).
19. Huang, Q., Wang, H., and Borisov, N. Privacy-Preserving Friends Troubleshooting Network. In *ISOC NDSS* (2005).
20. Hull, B., et al. Cartel: a distributed mobile sensor computing system. In *Proc. of SenSys* (2006).
21. Johnson, P. C., et al. Nymble: Anonymous ip-address blocking. In *Proc. of PET* (2007).
22. Lamport, L. Password authentication with insecure communication. *Communications of the ACM* (1981).
23. Mohan, P., Padmanabhan, V., and Ramjee, R. Nericell: Rich monitoring of road and traffic conditions using mobile smartphones. In *Proc. of SenSys* (2008).
24. Nambiar, A., and Wright, M. Salsa: A structured approach to large-scale anonymity. In *Proc. of CCS* (Nov 2006).
25. Perrig, A. The biba one-time signature and broadcast authentication protocol. In *Proc. of CCS* (2001).
26. Perrig, A., et al. Spins: Security protocols for sensor networks. *Wireless Networks* (2002).
27. Peterson, R. S., and Sirer, E. G. Antfarm: Efficient content distribution with managed swarms. In *Proc. of NSDI* (2009).
28. Przydatek, B., Song, D., and Perrig, A. SIA: Secure information aggregation in sensor networks, 2003.
29. Puttaswamy, K., Bhagwan, R., and Padmanabhan, V. Anonymity Preserving Data Aggregation using Anonygator. Tech. Rep. MSR-TR-2009-162, Microsoft Research, 2009.
30. Reed, M. G., Syverson, P. F., and Goldschlag, D. M. Anonymous connections and onion routing. *IEEE JSAC 16*, 4 (May 1998).
31. Rowstron, A., and Druschel, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *ACM Middleware* (2001).
32. Simpson, Jr., C. R., and Riley, G. F. NETI@home: A distributed approach to collecting end-to-end network performance measurements. In *PAM* (2004).
33. van Renesse, R., Birman, K., and Vogels, W. Astrolabe: A robust and scalable technology for distributed system monitoring, management and data mining. *ACM Transactions on Computer Systems* (2003).
34. Wagner, D. Resilient aggregation in sensor networks. In *ACM workshop on security of ad hoc and sensor networks* (2004).
35. Walfish, M., et al. Distributed quota enforcement for spam control. In *Proc. of NSDI* (2006).
36. Walsh, K., and Sirer, E. G. Experience With A Distributed Object Reputation System for Peer-to-Peer Filesharing. In *Proc. of NSDI* (2006).
37. Wang, H., Platt, J., Chen, Y., Zhang, R., and Wang, Y. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proc. of OSDI* (2004).
38. Yalagandula, P., and Dahlin, M. A scalable distributed information management system. In *SIGCOMM* (August 2004).
39. Zhu, B., Setia, S., and Jajodia, S. Providing witness anonymity in peer-to-peer systems. In *Proc. of CCS* (2006), ACM.