v-CAPS: A Confidentiality and Anonymity Preserving Routing Protocol for Content-Based Publish-Subscribe Networks

Amiya Kumar Maji and Saurabh Bagchi

Dependable Computing Systems Lab (DCSL)
School of Electrical and Computer Engineering
Purdue University, West Lafayette Indiana, USA
{amaji,sbagchi}@purdue.edu

Abstract. Content-based Publish-Subscribe (CBPS) is a widely used communication paradigm where publishers "publish" messages and a set of subscribers receive these messages based on their interests through filtering and routing by an intermediate set of brokers. CBPS has proven to be suitable for many-to-many communication offering flexibility and efficiency in communications between a dynamic set of publishers and subscribers. We are interested in using CBPS in healthcare settings to disseminate health-related information (drug interactions, diagnostic information on diseases) to large numbers of subscribers in a confidentiality-preserving manner. Confidentiality in CBPS requires that the message be hidden from brokers whereas the brokers need the message to compute routing decisions. Previous approaches to achieve these conflicting goals suffer from significant shortcomings—misrouting, lesser expressivity of subscriber interests, high execution time, and high message overhead. Our solution, titled *v*-CAPS, achieves the competing goals while avoiding the previous problems. In *v*-CAPS, the trusted publishers extract the routing information based on the message and the brokers keep minimal information needed to perform local routing. The routing information is cryptographically secured so that curious brokers or other subscribers cannot learn about the recipients. Our experiments show that *v*-CAPS has comparable end-to-end message latency to a baseline insecure CBPS system with unencrypted routing vectors. However, the cost of hiding the routing vectors from the brokers is significantly higher.

Keywords: content-based publish subscribe, privacy, anonymity, message latency.

1 Introduction

With the growing demand for adaptive and intelligent communication networks, content-based publish subscribe (CBPS) has gained significant attention in the research community over the last decade. Publish-subscribe in general is a communication technique whereby publishers "publish" messages and a set of subscribers receive these messages. This is more efficient than a publisher sending multiple point-to-point messages to each subscriber. Publish-subscribe offers a degree of decoupling between the publishers and the subscribers—a network of brokers together route the messages from a publisher to the correct set of subscribers. In traditional publish-subscribe systems, the subscribers express their interest in certain topics of messages and each message is published on one or more topics. Thus, conceptually, routing of messages to the subscribers is simple.

CBPS systems, which followed the development of traditional publish-subscribe systems, offer greater flexibility to the subscribers to express their interests. In CBPS systems, the subscriber defines a filter (a logical expression) on the content of a message, such as, a diabetic patient may be interested in availability of a drug named 'Glucotrol' where the store zip code is either '47901' or '47902' and unit price is less than '\$1'. Only messages matching the filter will be delivered to our hypothetical patient. Here the brokers execute more sophisticated algorithms for matching messages with constraints on attribute values in the filter (such as sub-string, equality, inequality). Typically, a hierarchy of brokers arranged in layers perform progressive filtering of the messages till they reach the correct set of subscribers. CBPS systems have seen significant research activity over the years resulting in excellent algorithms for filter matching, filter propagation through the broker network, and minimization of delivery latency [2], [4], [5], [6], [10]. These systems have also had mature industrial deployments [1], [2].

However, CBPS systems rely heavily on the integrity of brokers. Wang *et al.* [19] have shown that achieving message confidentiality, integrity, and auditability in the presence of malicious brokers is a challenging assignment. Consider the following scenario: Our hypothetical patient Jane is infected with HIV. She wants to subscribe for drug availability and preventive care newsletters for her disease from an online health information exchange that uses CBPS for content delivery. Due to the sensitivity of her disease, Jane doesn't want the brokers to learn about her subscription. Similarly, Dr. Watson, a publisher in the health information exchange, doesn't want to divulge contents of his messages to the brokers. But normal functioning of CBPS requires that the brokers should inspect both pieces of information (notifications and subscriptions) to route messages. We term the ensuing paradoxical problem—that of computing routing decisions based on encrypted notification and subscriptions—the *secure routing problem* (P_A). To further illustrate the complexity of this problem, let us assume that we have "magically" found a solution to P_A. However, to encrypt a notification, the publisher must know the precise set of subscribers that receive a notification and share a group key with them. This violates the publisher-subscriber decoupling property of CBPS. Furthermore, the set of subscribers is a function of the notification and may change with every message. We term this problem—that of dynamic group discovery and key exchange among publishers and subscribers—*dynamic subscriber group management problem* (P_B). An anonymity-preserving solution, like Tor [9], works well for single source to single destination, but not in the case when multiple patients need to subscribe to the same information, unbeknownst to others.

In this paper, we present v-CAPS, a routing protocol guaranteeing **C**onfidentiality of messages and **A**nonymity of subscribers in the presence of untrusted brokers in content-based **P**ublish-**S**ubscribe networks. In essence, our protocol solves the problem P_A mentioned in the previous paragraph. Our current work assumes a solution exists for the problem P_B so that for each notification, a key can be shared with a dynamically determined group of subscribers. Candidate solutions are available in [13]. A simplistic, but workable, solution will be to have a single key shared by each publisher with all the subscribers. Our brokers are curious in that they wish to inspect the messages and the recipients of messages, but are otherwise well-behaved in that they perform their routing decisions correctly. One can argue that the adversary model we consider is more insidious of the two—clear denial of service due to dropping the messages can be detected more easily. This class of privacy-preserving CBPS systems has been motivated by others in the literature [12], [15], [16].

Several researchers have tried to address P_A by using cryptographic techniques like computation on encrypted data [15], commutative encryption [16], or homomorphic encryption [12]. However, all of these approaches have their shortcomings—false positives or misrouting [15], lesser expressivity of subscriber interests or filters [12], [15], [16], high execution time [12], [15], and high message overhead [12], [15]. We make the important observation that routing in content-based publish-subscribe networks does not necessarily require inspection of the whole message. Instead, if a trusted publisher extracts the routing information from a message before encrypting it, then the problem reduces to hiding this information from malicious brokers. In our solution approach, the publisher looks at the commonality of interests among subscribers and encodes the routing information in the form of a routing vector (hence, the letter "v" in the name of our protocol). The routing vector (RV) is added to the header of a message and it allows brokers to compute their receiver lists. We present two versions of our protocol—one where the RV is left unencrypted (termed the RV protocol), and the second where the RV is further encrypted to achieve both confidentiality and anonymity (termed Secure RV or SRV protocol). Our simple approach eliminates the need for complex cryptographic operations, thereby, making it possible to incorporate the full generality of filters in baseline CBPS systems, with low computational overhead on the brokers. Our experimental results show that RV performs nearly as fast as a baseline CBPS in terms of latency. Achieving perfect anonymity (which we do through the SRV protocol), however, is significantly more costly and practical only for medium-sized networks. Unlike earlier approaches, the choice of encryption schemes is flexible in v-CAPS and continuing advances in faster content matching will render v-CAPS more efficient. For all practical purposes, v-CAPS does not have false positives (subject to the non-collision guarantees of cryptographic hash functions). The concessions that v-CAPS makes are added execution overhead at the publisher and some loss of decoupling between publishers and subscribers. However, the partial loss of decoupling has added advantage of auditability and enforcement of access control on subscriber interests.

The rest of the paper is organized as follows. Since a major portion of our protocol is described based on terminology used in Siena [4], a baseline CBPS system (i.e., without any privacy guarantee), we present the necessary background in Section 2. In Section 3, we highlight security goals, threat model, and assumptions in the proposed scheme. Section 4 presents the design of v-CAPS, guaranteeing message confidentiality. An enhanced protocol for incorporating subscriber anonymity is illustrated next. The protocol description is followed by an experimental evaluation of v-CAPS on a wide-area deployment. Finally, we discuss some unsolved design issues and conclude the paper.

2 Background

Content-Based Publish-Subscribe (CBPS) is an asynchronous communication paradigm where a message is routed based on its content instead of a fixed destination address. Typically, three types of nodes form the backbone of a CBPS network. These are – *publishers*, the entities that send a message into the network; *subscribers*, the entities that express their intention to receive messages with certain content; and *brokers*, the intermediate nodes that route messages from the publishers to the subscribers. Typically there are multiple levels of brokers between the publishers and the subscribers. CBPS has been shown to be an effective communication stratum under various scenarios — publishers and subscribers are linked transiently, fine-grained expression of interest can be made by subscribers, and some publishers and subscribers are ephemeral. It has been shown that CBPS is capable of delivering messages with low latency and of scaling to a large number of publishers and subscribers [2], [5]. The messages generated by publishers are termed as *notifications*. A notification consists of a collection of attributes and their values. Each element in this collection is a three-tuple <*attributeName*, *attributeValue*, *attributeType*>. E.g. a sample notification regarding available appointment schedule for Dr. Watson may look as follows:

```
wardName    cardiology          string
wardId      2131                integer
docName     Dr. Watson          string
totalSlots  20                  integer
apptSlots   list_of_slots       list_datetime
timeStamp   01/05/2011 09:00AM  datetime
```

The notification indicates that `Dr. Watson` in `cardiology` ward has 20 available appointment slots for the week (list for which is also given in the notification). The collection of attribute names and their data types define the *schema* of a notification.

A subscriber in a CBPS network may request for messages with certain attribute values. The interest of a subscriber is represented by a set of constraints over the attributes. Each *attribute constraint* is defined as a four tuple <*attributeName*, *operator*, *value*, *attributeType*>, where *operator* can be any boolean operator like $=$, $!=$, $>$, $<$, etc. or string operators like prefix, suffix, substring etc. For clarity of representation, we shall, however, denote attribute constraints as logical expressions in subsequent discussions and assume that the attribute type will be clear from the context. For example, an attribute constraint for the notification shown above may be `wardName="cardiology"`. A *filter* over a notification schema is defined as a conjunction of one or more attribute constraints, e.g., `(wardName="cardiology")`$\wedge$`(docName="Dr. Watson")` is a filter for receiving appointment slots for Dr. Watson in Cardiology ward. A subscriber may subscribe with one or more filters which are propagated from the subscriber to the publishers through the set of brokers. The notifications, on the other hand, are routed from the publishers to the matching subscribers through progressive filtering at different levels of brokers. We say that a notification **matches** a filter if all its attribute constraints are satisfied by the notification. Clearly, the filter `(wardName="cardiology")`$\wedge$`(docName="Dr. Watson")` matches the notification shown above.

Commonality between filters in the CBPS network is computed by a *covering relationship* as in [4]. We define that a filter F_1 **covers** a filter F_2, denoted $F_2 \prec F_1$ iff all the notifications that match F_2 also match F_1. For example, if F_1 = `(wardName="cardiology")` and F_2 = `(wardName="cardiology")` $\wedge$ `(docName="Dr. Watson")`, then $F_2 \prec F_1$. Loosely speaking, filter F_2 is less permissive, i.e. stricter, than filter F_1. Notice that, in the general case, notification sets of two filters may not overlap. Hence, the covering relation imposes a partial order on the set of filters. For efficient propagation of subscriptions and notifications, each broker in the CBPS network maintains two data structures—a *filter poset* and the *subscriber list* for each filter. The filter poset is the partially ordered set of filters received by a broker from its lower level brokers or subscribers, whereas, the subscriber list stores the set of subscribers for each filter.

2.1 Filter Posets

The filter posets, which denote partial ordering between filters at a broker, are represented as a collection of directed trees. In the tree an edge is drawn from filter F_1 to F_2 ($F_1 \rightarrow F_2$), iff $F_2 \prec F_1$. The root of a tree is termed as the *root filter*. Essentially, root filters are the set of filters that are not covered by any other filter. These trees may have overlap between themselves (i.e. they share some branches). However, the overall collection of trees form a directed acyclic graph (DAG). For notification forwarding decisions, each of the filters in the filter posets is associated with a set of recipients. A recipient may be either a subscriber or a next hop broker. For example, consider the simple network shown in Fig. 1. Here P is the publisher, B_1, B_2, B_3 are the brokers and S_1, S_2, S_3, S_4 are the subscribers with filters F_1, F_2, F_3, F_4 respectively.

We assume the covering relationship between filters to be $F_2 \prec F_1 \prec F_3$ while F_4 is independent w.r.t. other filters. The filter poset and the recipient list at each of the brokers in the network are displayed alongside each broker in Fig. 1.

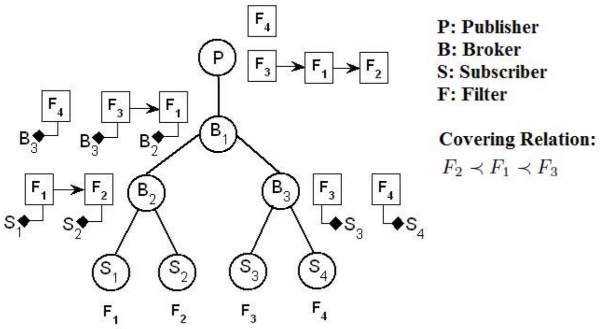


Fig. 1. An Example CBPS Network

The distinction between filter posets in *v*-CAPS and in baseline CBPS (Siena) lies in the content of each filter node. While each filter node in Siena contains a plaintext filter, filter nodes in *v*-CAPS store encrypted filters along with a unique filter ID. Additionally, each publisher in our protocol also stores the list of its filters and their covering relations in the form of filter posets. However, the publisher does *not* save recipient list for each filter. It is the responsibility of the brokers to maintain subscriber lists. The advantage of this design choice is that a publisher need not remember the topology of the network. It only remembers filters corresponding to the notifications it publishes. To avoid ambiguity, in further discussions of *v*-CAPS, we call the DAG representation of partially ordered list of filters at the publisher as the Publisher Filter Poset (PFPoset)and those at the brokers as Broker Filter Posets (BFPoset).

3 *v*-CAPS Basics

Our solution for confidentiality and anonymity preserving routing in CBPS networks, entitled *v*-CAPS, solves the *secure routing problem* (P_A) by introducing modified separation of duties for participating nodes (publishers, subscribers, brokers) and adding a level of indirection in filter matching. The protocol is built upon a typical publish-subscribe infrastructure, handles the full generality of baseline CBPS subscriptions, and does *not* require the presence of trusted third parties. The confidentiality and privacy goals of the proposed system are as follows:

1. *Notification Confidentiality*: No one except the publisher of a notification and its authorized subscribers can view the message content.
2. *Subscription Confidentiality*: No one except the subscriber and the publisher to whom it subscribes can know the content of a filter.
3. *Subscriber Anonymity*: A subscriber receiving notification N does not know other recipients (subscribers) of N.
4. *Filter Anonymity*: During routing, a broker can learn about matching between a notification and only those filters that are in its BFPoset. This ensures that the brokers have a very limited knowledge of which other brokers or subscribers receive a notification.

To satisfy the above security goals, we introduce two routing protocols named Routing Vector (RV) protocol and Secure Routing Vector (SRV) protocol. RV guarantees security goals (1) and (2). However, a resourceful attacker may be able to subvert goals (3) and (4) in RV. SRV, on the other hand, achieves all the security goals (1)–(4). We begin our protocol description in Section 4 by explaining RV and then highlight how we extend it to SRV in Section 5.

3.1 Threat Model and other Assumptions

We assume that the publishers and subscribers trust each other, but the subscribers do not trust each other. The brokers in the network may be malicious. We confine ourselves to an "honest but curious" model of the brokers. We assume that the brokers may try to learn the contents of a notification or subscription. It may also try to infer the mapping between a publisher and a subscriber. But the brokers follow the routing protocol correctly, i.e. it always forwards the notifications to the legitimate recipients as computed by the proposed scheme. We note that non-delivery of messages by malicious brokers can be easily detected by occasional rendezvous between publishers and subscribers. Appropriate legal actions may be taken to discourage such brokers. Similarly, notifications delivered by a malicious broker to illegitimate subscribers are unusable without the group key(s). However, "curiosity" of brokers leading to traffic analysis, etc. is challenging to detect and thwart. The threat model is, therefore, both practical and challenging. Earlier secure-CBPS schemes [11], [12], [15] are also built on this adversary model.

We assume that brokers in the CBPS network pre-compute a spanning tree connecting all the brokers and publishers. During subscription propagation, a filter is forwarded along the reverse edges of this spanning tree toward specific publishers. In case of a hierarchical broker network, the overlay network is equivalent to the spanning tree. Details of building a distributed spanning tree may be found in earlier work by Dalal and Metcalfe [8] and we omit the details in our protocol description.

3.2 Design Principles in v-CAPS

Our solution is motivated by two key observations. **First**, matching a notification against filters is several orders of magnitude faster in plaintext than matching on encrypted data [15], [12]. Therefore, it is desirable to compute filter matching against notifications in plaintext, rather than doing this at the brokers. **Second**, the brokers in baseline CBPS compute recipient lists of a notification based on a *match* that each broker computes. If the matching decision is added to a notification as a header, the untrusted brokers no longer need to inspect contents of notifications or filters to compute recipient lists.

4 *v*-CAPS Primitives

4.1 Subscribe

The subscription protocol allows subscribers to propagate their interests throughout the pub-sub network and to establish appropriate routes for receiving notifications. In our scheme, subscription consists of two stages. The first stage involves communication between the subscriber and the publisher and the second stage involves communication between the subscriber and the brokers. Details of the two stages are given below.

Stage I: *Contact Publisher* When a subscriber (S) joins the pub-sub network, it first registers itself with its preferred publisher(s) (P) through an auxiliary channel. Publisher verifies the identity of the subscriber and provides it with an authorization token. When the subscriber wants to receive notifications matching a given filter, it contacts the publisher with its authorization token and the filter (F). On receipt of F, the publisher computes as follows:

1. Does F exist in its PFPoset?

NO: (i) Assign a unique filter ID, ID_F to F; ii) Add F to its PFPoset; (iii) Compute its parents (F_{parent}) and children (F_{child}) sets

YES: (i) Lookup ID_F; (ii) Compute F_{parent} and F_{child}

2. Compute subscription token T_{sub} as:

T_{sub} `:= <parents>`F_{parent}`</parents><children>`F_{child}`</children>`
`<filter>`$ID_F|E_{k_s}(F)$`</filter>`

3. Send T_{sub} to the subscriber through an auxiliary point-to-point channel.

Note that E is any standard encryption function and k_s is the secret key used by a publisher to encrypt subscriptions and is known only to the publisher. The presence of $E_{k_s}(F)$ in the subscription request is not necessary for our content-based routing scheme. However, we store a copy of the encrypted filters at the brokers for the purpose of failure-recovery of the publisher.

Stage II: *Propagate Subscription* In this stage, the subscription token is propagated upstream through the broker network such that each broker updates its filter posets. After receiving T_{sub} from the publisher, the subscriber contacts the broker (B_s) it is connected to with T_{sub}. During subscription propagation, upon receipt of T_{sub} from a downstream node x_i, every broker B_i performs the following:

1. Does ID_F exist in its BFPoset?

NO: (i) Let, local parent list, $L_{parent} = nodes(BFPoset) \cap F_{parent}$, and local children list, $L_{child} = nodes(BFPoset) \cap F_{child}$; (ii) Add ID_F to BFPoset; (iii) Update parents and children edges of ID_F; (iv) Add x_i to recipient list of ID_F

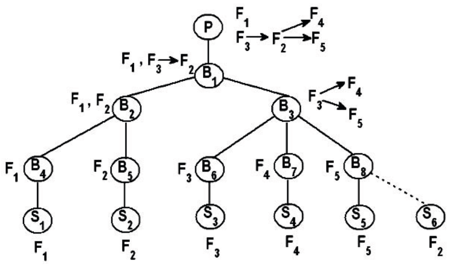
(v) If ($L_{parent} = \phi$)

Replace x_i in T_{sub} with B_i; Forward T_{sub} along the reverse edges of the spanning tree.

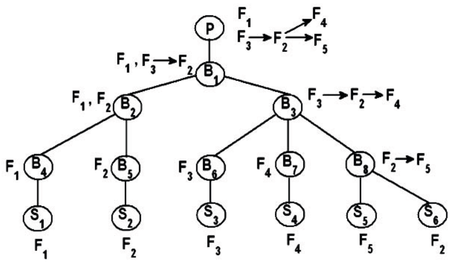
YES: (i) Add x_i to recipient list of ID_F

If the condition ($L_{parent} = \phi$) in step (v) above is not satisfied, it means B_i has already propagated a more general filter than F and T_{sub} is not forwarded.

On the other hand, satisfying this condition implies F is the root of some filter chain at B_i and it needs to be forwarded along the reverse edges of the spanning tree. Note that the addition of F at some broker may lead to *compaction of filters*, which we explain with the following example.


P
F_1
$F_3 \to F_2 \to F_5$, $F_2 \to F_4$
$F_1, F_3 \to F_2$
B_1
F_1, F_2
B_2
B_3
$F_3 \to F_4$, $F_3 \to F_5$
F_1 B_4
F_2 B_5
F_3 B_6
F_4 B_7
F_5 B_8
S_1 S_2 S_3 S_4 S_5 S_6
F_1 F_2 F_3 F_4 F_5 F_2

(a) Before joining of S_6



(b) After joining of S_6

Fig. 2. Subscription Forwarding: Before and After Joining of S_6

Consider the network in Fig. 2(a) consisting of a publisher (P), eight brokers (B_1–B_8), and six subscribers (S_1–S_6) with the given spanning tree. The filters for S_1 to S_5 are F_1–F_5 respectively. Now S_6 wants to subscribe with the filter F_2. The covering relation among the filters is assumed to be ($F_4 \prec F_2$), ($F_5 \prec F_2$), and ($F_2 \prec F_3$) (readers may use the filter graph beside P in Fig. 2(a) as a quick reference). By transitivity ($F_4 \prec F_3$), ($F_5 \prec F_3$). The filter posets at each of the brokers and the publisher before joining of S_6 are shown in Fig. 2(a). When P receives the request it computes $F2_{pred} = \{3\}$ and $F2_{succ} = \{4,5\}$ and includes these in a token T_{S_6} that it provides to S_6. When B_8 receives T_{S_6} it finds that it has no filter that belongs to $F2_{pred}$ but F_5 is in $F2_{succ}$. So it adds F_2 as a root filter and marks F_5 as its child. S_6 is added to the newly created recipient list of F_2. T_{S_6} is now propagated to B_3. Since $F2_{pred} = \{3\}$ and F_3 is already in B_3, T_{S_6} is not propagated any further. However, F_2 is inserted into the BFPoset at B_3. Both F_5 and F_2 at B_3 have B_8 as the recipient and ($F_5 \prec F_2$). This invokes a *compaction* of BFPoset of B_3. First, B_8 is removed from the recipient list of F_5 leaving it with no recipients. Therefore, F_5 is also removed from the filter poset at B_3. With this the routing path for S_6 is established. The final filter posets are shown in Fig. 2(b).

4.2 Publish

The publish protocol is initiated at the publisher. The publisher is responsible for extracting the *routing information* from a notification before sending it into the network. Hence, the publishers in v-CAPS first match a notification against the filters in PFPoset. The algorithm that we use for plaintext filter matching at publishers is the Siena Fast Forwarding (SFF) algorithm [5]. The function $M_{sff}(N, PFPoset) = F_{match}$ takes the plaintext notification N and $PFPoset$ as its inputs and produces a list of matching filter IDs (F_{match}). F_{match} is now added as a header to the notification and is termed RV. In the next step, the publisher with the help of *group manager*, computes the group key K_N for a notification N and encrypts the notification. The notification sent into the network by publisher looks as:

N_e =<RV>F_{match}</RV><Payload>$E_{K_N}(N)$</Payload>

The publisher now forwards N_e to broker B_1. The fact that the brokers do not have to do matching of filters against encrypted notifications allows us to avoid enormous performance penalties of computation on encrypted content. It may be argued that our scheme adds significant overhead on the publisher due to filter matching (as compared to baseline CBPS). But this is a practical approach considering publishers can be run on machines with sufficient computation power. Carzaniga et al. [5] have shown that even for a million subscriptions, plaintext filter matching typically takes time in the order of 10 ms on a desktop computer with 512MB of RAM. Our experiments also bear out this fact. Additionally, if filter matching is performed at each broker, this may lead to significant redundant computation as the brokers contain overlapping sets of filters.

4.3 Match

The Match() operation in v-CAPS is performed by brokers during notification delivery and its objective is to determine the list of receivers to forward the notification to. This operation is simplified by the fact that the publisher has already computed the RV and included it in the notification. The Match() operation at a broker B_i is done with the following simple steps:

```
Let receivers $R_{B_i} = \phi$
for each $ID_F$ in RV
    if ($ID_F \in BFPoset$) $R_{B_i} = R_{B_i} \cup receivers(ID_F)$
end for
```

B_i now forwards the encrypted notification to all the nodes in R_{B_i}. The brokers do not alter any part of the notification N_e and forwards an identical copy to all the recipients. Thus, for correct routing, a broker does not need to know either the content of a message or filters. Instead, routing may be performed using filter IDs generated by the publisher.

5 Secure Routing Vector (SRV) Protocol

The RV protocol presented in Section 4 achieves notification and subscription confidentiality (security goals 1 and 2) with the help of filter indirection and encryption. However, it does not guarantee security goals 3 (subscriber anonymity) and 4 (filter anonymity). Let us consider the following scenarios that may arise in the example in Figure 2(b):

1. S_2, a curious subscriber in our CBPS network, learns by external means that S_6 also receives notifications matching filter F_2. After receiving N_e from B_5, S_2 can easily identify S_6 as the other recipient of N_e. This violates *subscriber anonymity.*

2. B_2, a malicious broker learns by external means that B_3 subscribed to filter F_3. When, it receives N_e with `RV={1, 2, 3}`, it can easily identify B_3 as a recipient of N_e. Notice that filter F_3 is not even in the BFPoset of B_2. This violates *filter anonymity.*

From the examples above, let us now formulate the requirements of the SRV protocol. **First**, the RV should be encrypted in such a manner that, even if two notifications (N_1, N_2) both contain filter F_2 in the RV, it should generate different ciphertexts. This would help preserve subscriber anonymity. **Second**, the RV should be encrypted in such a manner that, a broker can only compute $\{BFPoset \cap RV_{enc}\}$. But it cannot learn which other filter IDs are in the encrypted RV_{enc}. This ensures filter anonymity. We adapt a prior solution on matching keywords in encrypted documents [18] to meet the last two requirements. The resultant solution is the *SRV protocol.* Before illustrating details of SRV, let us present a brief overview of the cryptographic technique in [18].

5.1 Background

Problem Statement: Assume, Alice has a set of secret documents D_1, D_2, .., D_k, where document D_i contains m_i words and every word is n bytes long. She encrypts these documents as Z_1, Z_2, .., Z_k and stores them on an untrusted file server Bob. Later, she wants to retrieve the documents containing an n-byte word w^*. However, Alice is reluctant to disclose either w^* or the encryption keys of Z_1, .., Z_k to Bob. So, Alice sends a query containing an encrypted keyword $x^* = E_{key}(w^*)$ to Bob. How can Bob find the precise set of encrypted documents $\mathbb{Z} = \{Z_i | D_i \text{ contains } w^*\}$ matching this query?
Solution: For clarity of representation, we abstract the encryption and match algorithms as a collection of functions. Interested readers may find the details of this algorithm in [18].

Let, $D_i = \{w_1.w_2....w_{m_i}\}$ (. denotes concatenation) is a plaintext document containing words w_1, w_2, .., w_{m_i}; w^* is a search word; and *keys* is a collection of secrets held by Alice (to be explained later). The secure search problem mentioned above can be solved by the following three functions:

- $\mathscr{E}(D_i, keys)$ is an encryption function that converts D_i to Z_i where $Z_i = \{c_1.c_2....c_{m_i}\}$ and c_j is a ciphertext for word w_j. $\mathscr{E}()$ can be used with different pseudorandom sequences to produce different encrypted versions of D_i for multiple encryptions.
- $\mathscr{F}(w^*, keys)$ is a cryptographic function that creates a search token $Q^* = \{x^*, k^*\}$ from w^*. Here, x^* is referred to as the *encrypted search word*, and k^* is referred to as the *search key* for w^*.
- $\mathscr{M}(Z_i, Q^*)$ is a match function which returns *true* **iff** w^* appears in D_i (using the above definition of Q^* which contains x^*, the encrypted keyword of w^*).

Internally, $\mathscr{E}()$, $\mathscr{F}()$, and $\mathscr{M}()$ use a standard encryption function (e.g. AES), a cryptographic hash function (e.g. SHA1), and a pseudorandom number generator as their building blocks.

Secrets Used: The algorithm uses three secrets, i.e., $keys = \{k_w, k', k_{seed}\}$. k_w is a secret key for AES, k' is a key for the cryptographic hash function, and k_{seed} is the seed for the pseudorandom number generator. All these secrets are stored by Alice and none of these is disclosed to Bob.

5.2 SRV Overview

In SRV, our trusted publishers are equivalent to Alice and the brokers to Bob. Let us first assign each of the filter nodes in PFPoset with a n-byte ID. Each notification N_i now contains n_i matching filter IDs in RV. Each RV can be considered as a document D_i that is n_i words long. We wish to restrict our brokers so that they can learn whether a filter ID_F appears in RV iff ID_F is in BFPoset. This can be achieved by encrypting RV using $\mathscr{E}$ and sharing the search token for filter ID_F with legitimate brokers. These search tokens are distributed during subscription stage of SRV. During notification forwarding, $\mathscr{M}$ is used to check the presence of filter ID_F in the encrypted RV. As in Section 5.1, publishers store the secret keys k_w, k', and k_{seed}. Let us now illustrate how we extend each of the primitives in RV for confidentiality and anonymity preserving routing. For brevity, we only highlight the additional steps needed in SRV.

5.3 Subscribe

Stage I: *Contact Publisher* After step 1 in RV, the publisher computes as follows (here the subscriber has subscribed with filter ID_F, which has parents in the PFPoset F_{parent} and children F_{child}):

2. Compute $F'_{parent} = \mathscr{E}(F_{parent}, keys)$ and $F'_{child} = \mathscr{E}(F_{child}, keys)$
3. Compute query token for filter ID_F as:
   $Q_{ID_F} = \mathscr{F}(ID_F, keys) = \{x_{ID_F}, k_{ID_F}\}$
4. Compute subscription token T'_{sub} as:
   T'_{sub} `:= <parents>`F'_{parent}`</parents><children>`F'_{child}`</children>`
   `<filter>`$x_{ID_F}|k_{ID_F}|E_{k_s}(F)$`</filter>`
5. Send T'_{sub} to the subscriber.

Note that the parents and children lists are encrypted to disallow the brokers from learning filter IDs that are not in their BFPoset. Since, subscribe is a one-time cost, the overhead of computing F'_{parent} and F'_{child} is not a performance bottleneck, however, it is essential for achieving security goals 3 and 4.

Stage II: *Propagate Subscription* BFPoset filter nodes in SRV contain $Q_{ID_F} = \{x_{ID_F}, k_{ID_F}\}$ instead of ID_F in RV protocol. Upon receipt of T'_{sub} from node n_i, every broker B_i computes as follows:

```
1. Does $x_{ID_F}$ exist in BFPoset?
  NO:  i) Compute local parent list
            $L'_{parent} = \{x_{ID_{F_i}} | \mathscr{M}(F'_{parent}, Q_{ID_{F_i}}) = true\}$
       ii) Compute local children list
            $L'_{child} = \{x_{ID_{F_i}} | \mathscr{M}(F'_{child}, Q_{ID_{F_i}}) = true\}$
       iii) Add $\{x_{ID_F}, k_{ID_F}\}$ to BFPoset
       iv) Update parent and children edges using $L'_{parent}$ and $L'_{child}$
       v) Follow step (iv) onwards as in RV (refer Section 4.1)
```

Note that the steps (i) and (ii) are necessary here since F'_{parent} and F'_{child} are encrypted.

5.4 Publish

To send a notification N into the network, a publisher first computes the matching filters $F_{match} = M_{sff}(N, PFPoset)$ as in RV. It then encrypts RV using $\mathscr{E}$ as:

$SRV = F'_{match} = \mathscr{E}(F_{match}, keys)$

The added overhead at the publisher, in comparison with RV, is the computation of $\mathscr{E}(F_{match}, keys)$. Our experimental results suggest that this overhead is only a small fraction of the end-to-end latency.

5.5 Match

Similar to RV, the objective of $Match()$ is to compute the recipient list for a notification N_e. However, encrypting the routing vector makes this operation significantly more complex compared to the RV protocol. Upon receipt of a notification N_e with $SRV = F'_{match}$, the broker first needs to compute the local match list L'_{match}, where

$L'_{match} = \{x_{ID_{F_i}} | x_{ID_{F_i}} \in BFPoset \texttt{ and } \mathscr{M}(F'_{match}, Q_{ID_{F_i}}) = true\}$

Hidden under the abstraction of $\mathscr{M}$, this is the most expensive part of our SRV protocol. The simple approach to compute L'_{match} would be to search for every filter in the BFPoset in every filter in the SRV. However, this would require a computation time of $m \times n$ matching operations, where m = number of filter IDs in F'_{match} and n = number of filter nodes in BFPoset. We reduce this cost by applying the following heuristics: (i) If a root filter with ID r_i does not match any entry in the SRV, i.e. $\mathscr{M}(F'_{match}, Q_{r_i}) = false$, then the broker does not do a search in the sub-tree of BFPoset rooted at r_i. This is based upon the observation that, if a message matches a certain filter F, then it must also match a root filter R above F. However, the performance gain of this optimization is dependent on the mix of filters in the network. If the covering between filters is high, then this heuristic would help. But, for a broker with lots of isolated filters (hence with a large number of root filters), this does not give significant improvement.
(ii) During computation of SRV match, if the broker observes that all its child brokers are already in the current receiver list, then it need not compute $Match()$ any further as all its child brokers must get that notification. This helps in reducing the SRV match times of the higher level brokers in a hierarchical broker network, which tend to have many filters and most often forward a message along all its downstream edges.

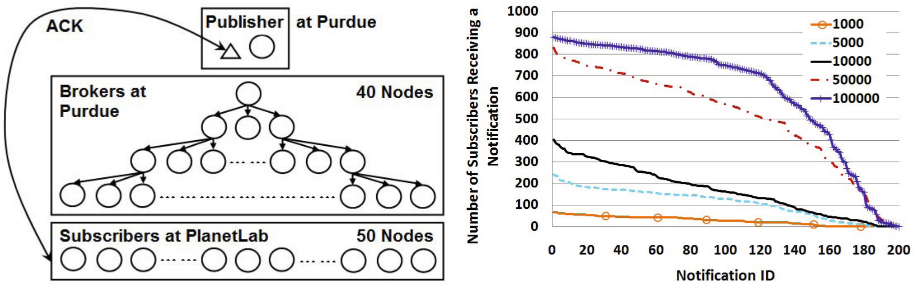
Once L'_{match} is computed by a broker B_i, the recipient list can be generated trivially. The controlled searching property of [18] ensures that a broker can only learn about the presence of its own filters in SRV and hence we are able to guarantee filter anonymity. Moreover, the encryption algorithm also ensures that if N_1 and N_2 both match filter F, the cipher IDs of ID_F in SRV_1, and SRV_2 are different. This helps us in protecting subscriber anonymity.

6 Experimental Results

We evaluated the performance of our protocols against baseline CBPS (Siena) with respect to end-to-end latency and computation time for notifications in a wide-area deployment. We implemented all the three protocols Baseline, RV, and SRV and deployed them on *PlanetLab* [14]—a worldwide computer systems testbed. Our experiments involve sending upto 100,000 subscription messages from the subscribers over a wide-area network, which, to the best of our knowledge, are the largest scale experiments on CBPS.

6.1 Experimental Setup

In our experiments, we created a hierarchical broker network with 4 levels (refer Fig. 3(a)). Each of the brokers in the top three levels were considered to have a fanout of 3, whereas, the leaf brokers were randomly connected to the subscribers. This constituted a broker network with 40 nodes. Each broker was hosted on a separate machine at Purdue University, 28 of which belonged to two clusters in our research group and the rest on public desktops in one of Purdue University laboratories. The reason for this choice of machines over PlanetLab machines is to reduce the sources of variability. We placed the less demanding subscriber processes on PlanetLab nodes.



(a) Layout of Nodes (b) Notification Popularity Distribution

Fig. 3. Experimental Setup and Workload Properties

At present, we generate notifications from a single publisher, with an interval of 3–5 seconds between successive notifications. The number of publishers can be easily increased with independent BFPoset data structures for each. The publisher was hosted on a desktop computer with 2GB RAM and 2.13GHz dual-core CPU running Ubuntu Linux 10.04. The subscribers were hosted on PlanetLab machines situated at widely varying geographical locations. We ran our experiments with upto 1000 subscribers hosted on 50 PlanetLab machines (i.e. 20 subscriber processes per machine). Each subscriber subscribed with 1–200 filters with a uniform random distribution, generating 100,000 subscriptions in our largest workload. In total, the experiments involved coordination between 1132 processes running on 91 machines over the Internet. All processes were run as user processes with default priority.

Workload Details. Since, there is no publicly available real-life workload for CBPS systems, we used *ssbg*, a component of the Siena software suite [3], to generate our workload. Attribute names for filters and notifications were chosen from a dictionary of 200 words. Each filter contained between 1 and 4 attribute constraints, while each notification contained between 1 and 9 attributes. For simplicity, all the attributes were defined to be of Integer type and their values were uniformly distributed in the range of $[1 - 100]$. Note that different subscribers may have overlapping subscriptions. We, henceforth, use the term *subscriptions* to define a set of filters from one subscriber, which may contain duplicate filters aggregated across all subscribers, and *filters* to define a set of unique filters. Using *ssbg*, we generated a total of five workloads with 100, 500, 1000, 5000, and 10000 filters respectively. Each subscriber now subscribed with a random subset of the filters, with uniform distribution in $[1-200]$. This generated the final workloads having 1000, 5000, 10000, 50000, and 100000 subscriptions respectively. Each workload contained 200 notifications. Due to the large number of subscriptions, this generated a significant number of notifications received by the subscribers (between 5459 and 125418 notifications at the subscriber end for the smallest and largest workloads respectively). Fig. 3(b) shows the popularity distribution of each notification. Based on the popularity distribution of notifications we classified them into three categories, namely, *popular*, *moderate*, and *esoteric*, where popular matches the most number of subscriptions and so on.

Latency Measurement. Latency, a simple concept in computer networks, is difficult to measure in wide-area networks. This is primarily due to coarse-grained clock synchronization accuracy over the Internet. In PlanetLab, we found that a large number of nodes had clock drifts in the range of seconds or even minutes. This compelled us to devise an alternative strategy for measuring end-to-end latency. In our experiments, prior to sending notifications, all the subscribers establish a dedicated connection to an acknowledgement server (ackServer) running on the publisher machine. After receiving a notification, the subscribers immediately forward an ACK with the notification ID and timestamp of notification. On receiving an ACK, the ackServer computes the total time spent by looking at the ACK timestamp. For future discussions, we define this closed-loop latency (from notification generation to receipt of ACK) as our *end-to-end latency.* To estimate the network RTT of PlanetLab nodes, the ackServer also periodically sends a timestamped packet to these nodes every 30 seconds which is reflected back by the subscriber nodes. We do not deduct $RTT/2$ from *end-to-end latency* as the network links were found to be asymmetric. Despite this, due to fluctuations of network latencies to and from PlanetLab nodes, we had to do some filtering of noisy points, where the estimated noise was greater than 5 ms.

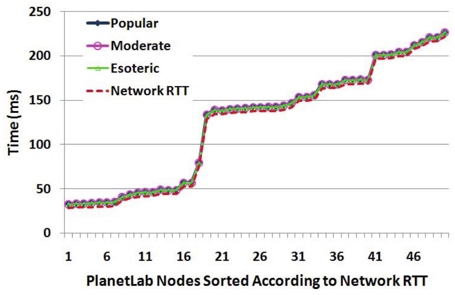
Other Implementation Details. The length of filter ID was chosen to be 16 bytes as this is also the block size of AES.We used an implementation of Siena Fast Forwarding algorithm [5], as the plaintext filter matching engine in all three protocols. For cryptographic operations we used the CryptoPP library [7] and built our networking code using C++ Sockets Library [17].

6.2 Evaluating Recurring Costs

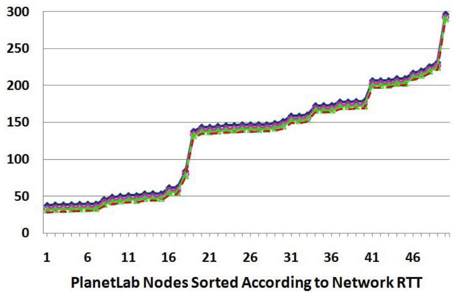
In v-CAPS, we try to achieve low computational cost for routing while guaranteeing confidentiality and anonymity. In this subsection, we present results for per notification cost.

End-to-end Latency. One of the crucial metrics for CBPS systems is to deliver notifications to the subscribers as fast as possible. In figures 4(a)–4(c), we present the end-to-end latency (as defined in 6.1) of different types of notifications for each of the protocols. The results are obtained from our experiments on the largest workload, i.e., 100,000 subscriptions. The X-axis represents various nodes on PlanetLab sorted according to their network RTT, while the Y-axis shows end-to-end latency in milliseconds. It can be seen from figures 4(a) and 4(b) that end-to-end latencies for baseline and RV are very similar. In baseline, end-to-end latency for all the nodes is within 5 ms of network RTT, whereas, in RV, this is within 10 ms of network RTT. There is no significant difference in end-to-end latencies with varying popularity type of notification. In baseline, all the three popularity types overlap on the same line and in RV, popular notifications have marginally higher latency than moderate and esoteric.

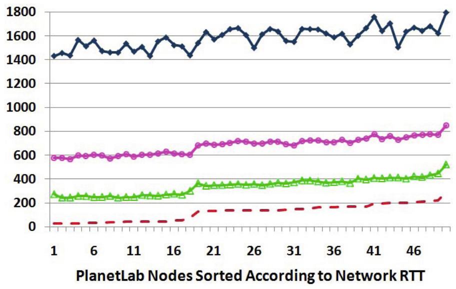
End-to-end latency in SRV, is however significantly higher than network RTT. This happens due to large matching time at the brokers. Latency also varies widely across popularity of notifications—popular notifications being the highest, followed by moderate and esoteric. The primary reason for this is the length of SRV for these categories. One of the key findings from our results is that achieving anonymity is significantly costlier than confidentiality alone. Our current implementation of RV has only marginally higher end-to-end latency than baseline. RV can be further improved by compressing the header and thereby reducing networking overhead due to increased size.



(a) Baseline



(b) RV

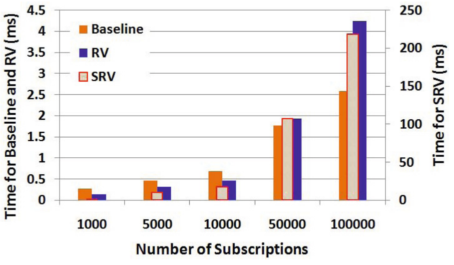


(c) SRV

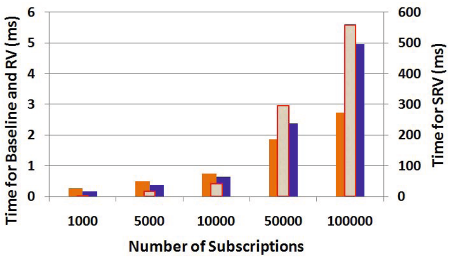
Fig. 4. End-to-end Latency of Notification Forwarding for Different Protocols with 100,000 Subscriptions

Computation Time. In figures 5(a)–5(c), we show the total computational cost of notifications with increasing workload size. The computational cost includes both cost at the publisher (for generating RV) and at the brokers (for matching). In each plot, we compare the performance of three protocols. *The large magnitude of SRV cost necessitated use of a secondary Y-axis.* We vary workload sizes along the X-axis, while the Y-axis represents time in ms. The bars for baseline and RV follow similar pattern across all popularity types. Even with the largest workload, the difference in computational time between RV and baseline is only 3 ms.

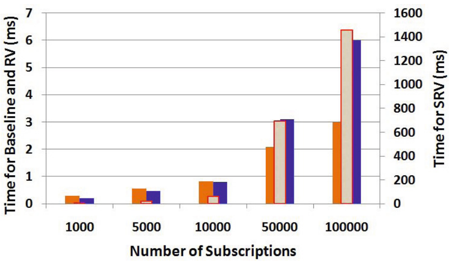
For SRV, computation time increases with increasing workload size as expected. However, the difference in computation time across popularity types is much more prominent (~1500 ms for popular, ~560 ms for moderate, and ~220 ms for esoteric in our largest workload). This is because the number of matching filters is significantly different in the three categories (popular > moderate > esoteric) and the cost of processing at a broker for each filter is high in SRV. Contrary to a possible criticism, the cost at the publisher is quite low. Though this constituted the largest fraction of overall computation time in RV, it is only marginally higher than the computation time in baseline (within 1.5 ms) even for 100,000 subscriptions. For SRV, the cost at the publisher (~4 ms for 100,000) is three orders of magnitude lower than the cost at brokers (~1500 ms for popular at 100,000).



(a) Esoteric



(b) Moderate



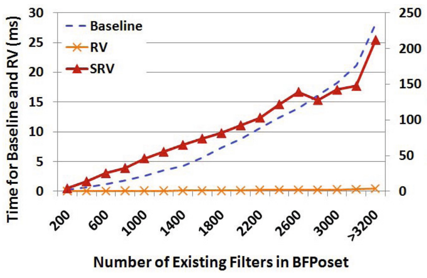
(c) Popular

Fig. 5. Computational Cost for Notifications with Increasing Number of Subscriptions. Note the use of two separate Y-axes due to widely varying values–the left axis for baseline and RV and the right axis for SRV.

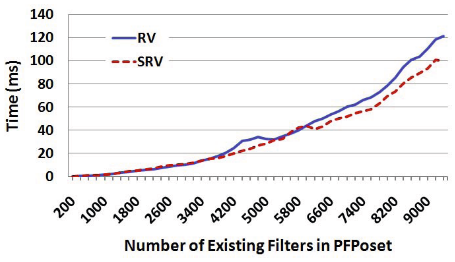
6.3 Evaluating One-time Costs

In this section, we evaluate the cost of registering a new subscription at publishers and brokers. At the publisher, cost of a new subscription amounts to adding the filter in PFPoset, reorganizing PFPoset edges, and computing encrypted parent and children lists. At brokers, this is the cost of evaluating local parent and children lists, reorganizing BFPoset edges, and in some cases, propagating the subscription up the broker hierarchy. Notice that since subscriptions from different subscribers may contain duplication of the same filter(s), majority of the subscriptions involve only a lookup operation at publisher and brokers incurring very small overhead (in the order of a hundredth of a millisecond). In figures 6(a) and 6(b), we consider cost of adding *new* filters only. The X-axis in both the figures represent number of filters already existing at a broker or a publisher. We grouped this into buckets of size 200, i.e. when number of existing filters is (0, 200], (200, 400], etc. The point 400 represents the range (200, 400] and so on. The Y-axis represents computation time for adding a new filter in milliseconds. For brokers, the number of *filters* at a broker was upto 3400 while at the publisher it was 9400 for workload size of 100,000 subscriptions.

It can be seen from Fig. 6(a) that the cost of adding a *new* filter at a broker is much higher in SRV (~230ms for the largest workload). This is insignificant in RV (< 1ms), since, subscription propagation in RV involves only lookup operations. For baseline the cost was ~25ms for our largest workload. The RV cost is lower because the publisher has already done the processing to figure out which will be the parent and children filter nodes in the BFPoset, while in the baseline, the broker has to compute this. The slow subscription processing in SRV may not be a severe bottleneck because this is a one-time cost incurred only when a filter is entered for the first time and over time, most subscriptions result in duplicate filters.



(a) Cost at Brokers



(b) Cost at Publisher

Fig. 6. Cost for Adding a New Filter at Brokers and Publishers

Fig. 6(b) shows the cost of adding a new filter in RV and SRV with increasing number of filters in PFPoset. The cost in SRV(RV) reaches upto 100(120) ms for 9200 filters. To reduce this cost, the publisher may also pre-load a set of known subscriptions in its PFPoset. The costs for RV and SRV are comparable because the same processing happens at the publisher; the slight differences (in fact SRV is faster) is explained by the different orders in which filters arrive at the publishers and the fact that different mix of existing filters affect processing time.

6.4 Message Overhead

During notification forwarding, we measured the length of RV(SRV) in each notification, where RV length is defined as the number of filterIDs in RV. This gives us an estimate on the message overhead of v-CAPS over baseline CBPS. Our experimental results are presented in Fig. 7. Since both RV and SRV protocol have identical header lengths, we show only one plot for both of them. The X-axis here represents various workload sizes. Since our experiments involved synthetic payloads for each notification, comparing the header with the payload as a measure of overhead will be misleading. We, therefore, normalized the total header size (Num notifications$\times$RV or SRV Length$\times$16 bytes) by the total number of subscribers receiving each notification. This represents the average number of additional bytes spent per subscriber for a given class of notification. This cost is displayed by the line plots in Fig. 7. It is encouraging to find that the cost per subscriber is less than 4 bytes in all cases. One may argue that for a small notification with lots of non-overlapping subscriptions (i.e. with a long RV) message overhead is substantial. This cost is indispensable since filter IDs in v-CAPS are equivalent to virtual "destination addresses." A possible improvement would be to add filter coverage information in the header (RV), so that, during notification propagation, the brokers only forward relevant portions of the RV to lower level brokers.

7 Related Work

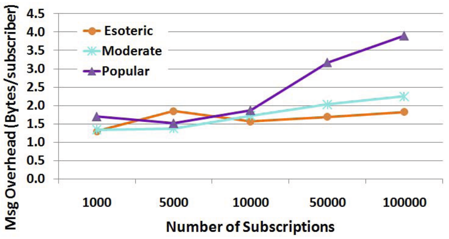


Fig. 7. Message Overhead for Routing in RV(SRV) with Increasing Workload Size

Over the past decade and a half, publish-subscribe has been extensively studied as an efficient model of communication. Security in CBPS systems have been achieved under different settings—different network topologies, varying degrees of trust between the communicating entities, and varying flexibility of subscription predicates. Another significant problem—that of key management in such a dynamic environment has been studied in [13]. Majority of these approaches have the goal of securing CBPS against the vulnerability of malicious brokers. They balance this source of vulnerability against the common CBPS design in which brokers need to examine the content of messages to make routing decisions. In [15], the authors adapted the schemes presented in [18] and other techniques on computation on encrypted data. They build a confidentiality-preserving CBPS system that supports equality, inequality, and range matches for numeric attributes, and keyword searches for strings. The experimental results show that for 1,000 subscriptions, compared to the corresponding insecure operations, equality is 6 times more expensive, inequality is 1.7-3.0 times more expensive, and range matching is 6 times more expensive. Apart from the computation cost, this scheme suffers from three other significant drawbacks—restrictive filters on subscriber interests, high communication overhead (the encrypted message is 15 times the size of the plaintext message), and false positives for filter matching. On the other hand, [12] addresses privacy in CBPS by applying Paillier homomorphic encryption for equality, and inequality matches on numeric attributes. This scheme performs filter matching with two primitives—1) *blinding* the attribute value for subscriptions and notifications at the publisher, and 2) *matching* the blinded values in filters and notifications. The authors have not presented any implementation of their protocol in a publish-subscribe system. However, standalone experimental results for the cryptographic operations show that blinding **one** attribute value at the publisher takes 10-15 ms for a key length of 1024 bits and matching between one blinded attribute value and one blinded constraint takes 100μs. Apart from these, complexity of key management and large message size are other drawbacks of this solution. Another research work, presented by Molva *et al.*, try to achieve confidential routing in CBPS using multiple layer commutative encryption (MLCE). The protocol computes matching between a filter and a notification by comparing their encrypted strings. As a result, this solution is limited to equality matches. Moreover, for a k-layer commutative encryption, a broker would be required to know the sender or recipient of a message at distance k from itself. Similarly, while propagating the messages downstream, a broker would be required to encrypt it separately with each of the recipients' (at distance k) keys. Ion *et al.* [11] used Attribute-based Encryption (ABE) and multi-user Searchable Data Encryption (SDE) to achieve confidentiality of notifications and filters without losing any decoupling property of CBPS. Their scheme, however, needs the presence of a trusted authority which our solution does not. Due to absence of experimental results in [11] we cannot compare the performance of this scheme with ours.

8 Discussion and Future Work

Our design and implementation of *v*-CAPS shows that it is possible to support privacy in CBPS. In other words, it is possible to handle the balance between the need to route by the brokers, and that the brokers are not trusted and may be curious. We achieve this without sacrificing the generality of filters in baseline CBPS. Further, we achieve this with acceptable overhead (in terms of time) over the baseline CBPS, *if* one is willing to accept a slight risk of the broker getting to know which other brokers will see a message, which may happen with the unencrypted routing vectors. However, if we want to eliminate this risk, we have to perform matching of encrypted filters with encrypted routing vectors, which is significantly more costly—for 10,000 filters, the end-to-end latency is just under 1.5 sec. However, we believe that there are two promising directions to resolve this problem. First, the matching algorithm at the broker can be easily parallelized—each entry in the secure routing vector (SRV) is matched in parallel. Thus, brokers running on multi-core machines can leverage this. Second, the publishers can send a hierarchical SRV, which will allow the broker to perform the simple optimization that if a routing vector does not match a filter, it will not match any filter in the sub-tree rooted at that filter.

Our work has not addressed the issue of fault tolerance, either at the brokers or at the publishers. A practical system needs to handle crash failures of both entities. Fault tolerance for broker failures is orthogonal to the privacy requirement of CBPS and we believe *v*-CAPS can be easily applied to a baseline CBPS that has redundancy to deal with broker failures. To handle publisher failures, the system needs to be able to recreate the publisher filter poset. This information is conceptually contained in the union of the filter posets at all the brokers. The challenge will be in gathering them in an efficient manner. The leaf brokers can potentially violate the anonymity requirement of the subscribers because they are directly forwarding messages to the subscribers. Hence, to achieve subscriber anonymity from leaf brokers, it will be required to interpose an anonymizing network between these entities. This anonymizing network will have no notion of the filters of the subscribers.

References

1. Barnett, D.: Publish-subscribe model connects tokyo highways. Web article, `http://www.industrial-embedded.com/articles/barnett/`
2. Bhola, S., Strom, R.E., Bagchi, S., Zhao, Y., Auerbach, J.S.: Exactly-once delivery in a content-based publish-subscribe system. In: DSN 2002: Proceedings of the 2002 International Conference on Dependable Systems and Networks, pp. 7–16 (2002)
3. Carzaniga, A.: Siena download. Web article, `http://www.inf.usi.ch/carzaniga/siena/forwarding/index.html`
4. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems 19(3), 332–383 (2001)
5. Carzaniga, A., Wolf, A.L.: Forwarding in a content-based network. In: Proceedings of ACM SIGCOMM 2003, Karlsruhe, Germany, pp. 163–174 (August 2003)
6. Chandramouli, B., Yang, J., Agarwal, P.K., Yu, A., Zheng, Y.: Prosem: scalable wide-area publish/subscribe. In: SIGMOD 2008: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 1315–1318 (2008)
7. Crypto++ library - a free c++ class library of cryptographic schemes. Web article, `http://www.cryptopp.com/`
8. Dalal, Y.K., Metcalfe, R.M.: Reverse path forwarding of broadcast packets. Communications of the ACM 21(12), 1040–1048 (1978)
9. Dingledine, R., Mathewson, N., Syverson, P.: Tor: the second-generation onion router. In: Proceedings of the 13th Conference on USENIX Security Symposium, SSYM 2004, vol. 13, p. 21 (2004)
10. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys 35(2), 114–131 (2003)
11. Ion, M., Russello, G., Crispo, B.: Supporting Publication and Subscription Confidentiality in Pub/Sub Networks. In: Jajodia, S., Zhou, J. (eds.) SecureComm 2010. LNICST, vol. 50, pp. 272–289. Springer, Heidelberg (2010)
12. Nabeel, M., Ning Shang, E.B.: Privacy-preserving filtering and covering in content-based publish subscribe systems. Tech. rep., Purdue University (June 2009)

13. Opyrchal, L., Prakash, A.: Secure distribution of events in content-based publish subscribe systems. In: SSYM 2001: Proceedings of the 10th Conference on USENIX Security Symposium, p. 21 (2001)
14. Planetlab: An open platform for developing, deploying, and accessing planetary-scale services. Web article, http://www.planet-lab.org/
15. Raiciu, C., Rosenblum, D.S.: Enabling confidentiality in content-based publish/subscribe infrastructures. In: Securecomm and Workshops 2006, August 28-September 1, pp. 1–11 (2006)
16. Shikfa, A., Önen, M., Molva, R.: Privacy-Preserving Content-Based Publish/Subscribe Networks. In: Gritzalis, D., Lopez, J. (eds.) SEC 2009. IFIP AICT, vol. 297, pp. 270–282. Springer, Heidelberg (2009)
17. C++ sockets library: A class library wrapping the berkeley sockets c api. Web article, http://www.alhem.net/Sockets/index.html
18. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 44–55 (2000)
19. Wang, C., Carzaniga, A., Evans, D., Wolf, A.: Security issues and requirements for internet-scale publish-subscribe systems. In: HICSS 2002: Proceedings of the 35th Annual Hawaii International Conference on System Sciences, pp. 3940–3947 (January 2002)