A Private, Secure, and User-Centric Information Exposure Model for Service Discovery Protocols

Feng Zhu, *Student Member*, *IEEE*, Matt W. Mutka, *Senior Member*, *IEEE*, and Lionel M. Ni, *Fellow*, *IEEE*

Abstract—Service Discovery as an essential element in pervasive computing environments is widely accepted. Much research on service discovery has been conducted, but privacy and security have been ignored and may be sacrificed. While it is essential that legitimate users should be able to discover services, it is also necessary that services be hidden from illegitimate users. Since service information, service provider's information, service requests, user presence information, and user's identities may be sensitive, we may want to keep them private during service discovery processes. There appears to be no existing service discovery protocols that solve these problems. We present a user-centric model, called PrudentExposure, which exposes minimal information privately and securely. Users and service owners exchange code words in an efficient and scalable form to establish mutual trust. Based on the trust, secure service discovery sessions are set up. The model is further improved to counter attacks. We analyze the mathematical properties of our model, formally verify our security protocol, and measure the performance of our prototype system.

Index Terms—Pervasive computing, privacy, security.

1 Introduction

As we move toward pervasive computing environments with various computational devices and services embedded in the surroundings of our daily life, service discovery as an essential element to access network services (services for short) is widely accepted. Many products and standards have emerged and much research work has been conducted. Privacy and security, however, have been ignored and, therefore, may be sacrificed when services may be discovered or used by any user. In this paper, we present a secure, efficient, and scalable model, called *PrudentExposure*, to allow legitimate users to discover and use services easily, while it excludes others from seeing sensitive information within pervasive computing environments.

To help address the problems and solutions, we sketch a scenario of Bob discovering services at different places. Bob's house has various wired and wireless computing devices. He shares these devices and services with his family members. As usual, he puts his cell phone, PDA, and MP3 player in his handbag and a Bluetooth earphone in his pocket, and then travels to his office. On the way to his office, he may wear his Bluetooth earphone and use it to discover his Bluetooth MP3 player and listen to songs. Nevertheless, he does not want others to know what is in his bag. In his office, he uses his computer, cell phone, and MP3 player. When he goes to Alice's office, they look at a document on the office file server simultaneously with their respective laptops via the office's wireless LAN. The devices within his pocket, however, should not be able to discover and use Alice's personal services on the devices in her purse, and vice versa, unless Alice later provides a user name and a password for him to access.

Envision that within pervasive environments, dozens to hundreds of devices and services may surround a user. Over time, she or he may utilize thousands of services at different places. Meanwhile, the user may be the owner of some services. When discovering services in such environments, much information is sensitive and should be exposed with prudence. We identified that the following information is sensitive during service discovery. Keeping it private is our goal:

- At the service owners' side, service information, owners' identities, and presence information should only be exposed to legitimate users and hidden from others.
- At the users' side, identities used for authentication, user's presence information, and service query information should only be exposed when it is necessary.

Achieving the two goals is challenging in pervasive computing environments. First, services coexisting in a place may belong to different owners. For example, in Bob's work place, services belong to Bob, Alice, or the office. Second, user mobility and service mobility cause the available domains and services to change dramatically. For instance, different services are available in Bob's house and on his way to the office. Bob carries his mobile devices and the services on them move with Bob. Third, since users

- *F. Zhu and M.W. Mutka are with the Department of Computer Science and Engineering, 3115 Engineering Building, Michigan State University, East Lansing, MI 48824. E-mail: {zhufeng, mutka}@cse.msu.edu.*
- *L.M. Ni is with the Department of Computer Science, The Hong Kong University of Science & Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: ni@cs.ust.hk.*

Manuscript received 2 Mar. 2004; revised 1 Aug. 2004; accepted 9 Dec. 2004; published online 15 Feb. 2006.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number TMC-0043-0304.

act in many different roles, they use different identities for different administrative domains. For instance, Bob uses one user name to access his office PC and another user name for his PDA. These three issues make pervasive service discovery more challenging:

- *Requiring a priori knowledge of services and their relative domains.* If a user does not know the existence of a service in advance, for example, a new service, he may not be able to input the relative identities for authentication. Similarly, if a user does not know to which domain a service belongs, he again misses the opportunity to use the service.
- *Usability problem.* Usability is very poor when a user has to remember the relation among domains, authentication identities, and pervasive services. In addition, it is distracting for users to actively identify existing domains before each service discovery.

Many service discovery protocols have been proposed. Nevertheless, to the best of our knowledge, there is no service discovery protocol that meets the goals and the new challenges for pervasive computing environments as we discussed above. Most protocols do not consider privacy and security issues [1]. Devices and services are permitted to discover each other freely. Therefore, services may be used by anyone and, thus, access control is violated. Furthermore, by exposing service and domain information in an unrestricted manner, services are inclined to be attacked and privacy may be sacrificed. Additionally, the user's or owner's presence information is exposed. For example, if Alice's MP3 player announces its existence every five minutes, her itinerary is known. Secure service discovery protocols, such as Secure Service Discovery Service (SSDS) [2], manage services centrally. In pervasive computing environments, user and service information exposed to central servers may not be appropriate. For instance, Bob may not want to register his MP3 player with the office's directory. Universal Plug and Play (UPnP) Security [3] provides many authentication and authorization mechanisms, but the generic mechanism of automatically selecting services is limited to one's own services. The mechanism requires every device to reply and, thus, service privacy is sacrificed. Moreover, an important issue that has not been addressed in existing protocols is how to properly acquire user identities for service discovery. Heterogeneous devices may potentially be used to discover various services, which may require different authentication mechanisms. It seems infeasible to support all mechanisms on every device for authentication.

Our service discovery architecture is user-centric; user identities are managed centrally and are supplied automatically. Hence, users are free from the burden of associating identities with domains and pervasive services. If users have privileges, they do not need a priori knowledge of the existence of services or the knowledge of which domain a service belongs. Moreover, devices are free from authenticating users. To achieve our goals, we expose information prudently. Sensitive information is protected throughout the service discovery process. Identities and service information are exchanged in a secure and private form, more specifically within the Bloom filter form [4]. Further information exposure between a discovering party and a service provider is based on mutual trust. Moreover, devices and services are less likely attacked when we protect their sensitive information by hiding it and by not responding to arbitrary queries in the first place.

Our model is complete such that users do not miss any services that they should discover and services let all legitimate users discover them. Our protocol is formally verified by using and extending BAN logic [5]. Compared to existing protocols, our protocol introduces little overhead. The rest of the paper is as follows: In Section 2, we discuss work related to secure and private service discovery protocols. In Section 3, we present our user-centric service discovery architecture. Next, in Section 4, we illustrate our PrudentExposure model. Then, we analyze and evaluate our model in Section 5. Last, in Section 6, we conclude and discuss our future work.

2 Related Work

Active service discovery research has occurred in both industry and academia. Major operating system vendors have shipped service discovery protocols with their operating system products, such as Sun Microsystems's Jini [6], Microsoft's UPnP [7], and Apple Computer's Rendezvous [8]. Several organizations have also proposed service discovery protocols such as Bluetooth Service Discovery Protocol [9] from the Bluetooth Special Interest Group, Salutation from Salutation Consortium [10], and Service Location Protocol (SLP) Version 2 by IETF [11]. Some representative academic projects are DEAPspace [12], Intentional Naming System (INS) [13], and INS/Twine [14]. A detailed comparison of these protocols may be found in [1].

These protocols may be roughly classified into two models: *client-service* and *client-service-directory*. In the client-service model, clients (discovering devices) first inquire about the services' availability. After examining the client's queries, matched services return replies. Among the replied services, clients select and contact services. Since all services are involved in a query, the model is not scalable. To support thousands of computing services, directories may be used to store service information. In the client-service-directory model, a client discovers a directory and then queries the directory for service information. After receiving a list of matched services in the directory's reply message, the client chooses and contacts a service. Services, on the other hand, discover and register service information with directories.

Most service discovery protocols, however, do not provide security and privacy support. Therefore, any services may be discovered and used by any user via service discovery protocols. Secure service discovery protocols, such as SSDS [2], UPnP Security [3], and a secure Jini service discovery in [15], have proposed models or solutions for different environments with different security requirements. Nevertheless, these protocols may only be suitable for single administrative domain environments, such as enterprise environments or home environments, in which services are owned by the same owners and administrated centrally.

As one of the first secure service discovery protocols, SSDS has many built-in security features including authentication, authorization, data and service privacy, and integrity. In SSDS, directories, known as Service Discovery Service servers, are trusted. Clients and services authenticate with the directories for service lookups and announcements, respectively. SSDS is good for enterprise environments, where users are willing to expose identities and service requests, and where services are willing to expose their service information to central directories. In pervasive computing environments, however, trusted central directories may not be appropriate. Besides the privacy problem, access control is difficult to enforce, manage, and maintain. The situation of multiple directories from different owners coexisting is not addressed.

UPnP Security [3] offers many security features. In UPnP Security, devices are named personally and identified by public keys. Different authorization methods are supported, for instance, access control lists, authorization servers, authorization certificates, and group definition certificates. UPnP Security provides a generic method to differentiate an owner's devices from others. Based on the hash of a device's public key, a user can decide whether he owns the device or not. The approach is inefficient and lacks privacy for services. In our scenario, if Bob waits in a train station and wants to use his earphone to discover his MP3 player, then all MP3 players in the vicinity will reply to the service discovery request. Although Bob's earphone will tell which MP3 player is his, he may not have privileges to access many of the MP3 players and should not even be aware of the existence of the services.

Other work also influences our approach. We borrow the ideas of using public keys and hashes of public keys to represent domains from SDSI [16] and SPKI [17]. We use Bloom filters [4] extensively in our protocol because using Bloom filters provides a scalable method for membership tests. In Summary Cache [18], cached Web pages at a proxy are represented as a Bloom filter form and shared among neighbor proxies. To support wide-area service discovery in SSDS [2], service information is in the Bloom filter form when building the hierarchical directory structure. The Resurrecting Duckling security policy [19], [20] provides a new way for authentication in pervasive computing environments. By mimicking the behavior of mother ducks and ducklings, the policy sets up a master-slave relation. We borrow this idea to associate owners' devices and services with their directories.

3 A New Service Discovery Architecture

Based on the client-service-directory model (see Section 2), our architecture has four types of components—clients, services, directories, and user agents. Clients and services are similar to their counterparts in the client-service-directory model. A client is a device a user utilizes to access services, for example, Bob's Bluetooth earphone in our scenario. A service is a networked computing resource such as Bob's Bluetooth MP3 player. (A device may change its role; for example, if Bob downloads new songs to his MP3 player from his PC, the MP3 player becomes a client in that context.) Directories store and dynamically update service information. However, they have tight relations with the services (see Section 3.2). We introduce a new component, called a user agent, to facilitate users managing identities for authentication.

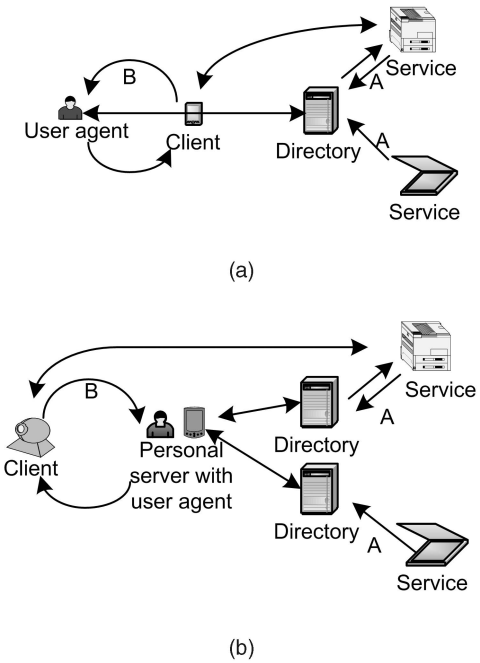

Fig. 1. Two service discovery examples. (a) A PDA (client) discovering services. (b) A digital camera without a public key operation capability discovering services via a user agent on a Personal Server [21] (a powerful processing and networking handheld device).

The four components are classified as: a service discovering party, which consists of a user agent and a client, and a service provider, which consists of services and a directory. Two discovery diagrams are shown in Fig. 1. There are three pairs of relationships among the four types of components: a relationship within a service provider (binding A in Fig. 1), a relationship within a service discovering party (binding B in Fig. 1), and a relationship between a service provider and a service discovering party. We discuss the first two relationships in Sections 3.2 and 3.3, respectively. The last relationship is discussed in Section 4 when we discuss the detailed discovery process.

3.1 Target Environments

Our model targets pervasive computing environments in which users discover services in their vicinities. Our prototype system uses IEEE 802.11b and UDP multicast, but our model may be deployed to wired and wireless LAN (such as IEEE 802.11x and Bluetooth) or other communication mechanisms that support broadcast or multicast. If a device has multiple interfaces, it may simultaneously communicate via all interfaces to discover services. In a place, multiple administrative domains may coexist. Service owners (owners for short) or their administrators manage their domains, respectively. For example, in Bob's office, Bob manages his personal devices and services, Alice manages hers, and system administrators manage the office's computing resources. Moreover, users utilize different roles to access services. To access the office's services, Bob uses a user name and password pair; to access

Alice's services, Bob uses another user name and password pair that he acquires from Alice.

Our model does not require a fixed underlying authentication and authorization mechanism. However, we assume that users have valid identities from service providers. A user may use a user name and a password to access one service and use a certificate to access another service.

3.2 Owner-Based Service Management

In insecure service discovery protocols, services announce their information periodically or after directories' solicitations, and directories accept all service registrations. Exposing service information to everyone sacrifices privacy and may cause attacks. In the meantime, directories may be swamped with unrelated service information and degrade their performance. Nevertheless, deploying centralized secure service discovery protocols in pervasive computing environments causes new problems. If an owner registers his services with a central directory, he not only exposes his service information to the directory but also loses control of who should acquire the service information. Moreover, in the client-service-directory model, directories announce their existence information periodically or when clients or services discover directories. In pervasive computing environments, if a directory represents a person, the announcements expose the person's presence information.

To solve the problems, service registration in our model is selective and owner-based. A directory only stores information of the services that belong to the same owner. Similarly, a service only announces its information to the directory of the same owner. For example, in our scenario, in Bob's office, services owned by Bob register with Bob's directory and services owned by the office register with the office's directory. Furthermore, Bob's directory does not accept Alice's service registrations. Thus, Bob's services are kept private and service information is exposed under the control of his directory and in turn under his control. In addition, services and directories use soft state and a lease-based service registration mechanism (used in many service discovery protocols [1]) to maintain the freshness of the service information.

In our model, directories do not announce their information periodically. When hearing a discovery message, a directory checks whether the discovery message is sent by a valid user or service. If the check returns a positive answer, the directory replies back. Otherwise, the directory keeps silent. The detail discovery processes are discussed in Section 4.

Besides the service registration, a directory and a service have a long-term control relationship: The directory controls the service. This relationship only needs to be set up once. For instance, when the owner first acquires the service, he associates the service with his directory. We borrow the master-slave relationship idea from the Resurrecting Duckling approach [19], [20]. A directory is a mother duck (master) and services are ducklings (slaves). After a duckling (service) accepts an imprinting from a mother duck (directory), a secure communication channel is established. Moreover, only the directory can instruct the service to break this control relationship.

Directories authenticate users and maintain access control lists. All service accesses need to go through the directories to get permissions. When a client is authorized to access a service, a directory informs the service specific policies. Owners (or administrators) can enforce access control and manage user privileges in one place: the directories. By moving the authentication and authorization from services on to the directories, security and privacy requirements on services are dramatically reduced. Managing user accounts and privileges do not need to be done on services individually for heterogeneous devices. Revocation of a user's privilege can be easily done on the directories. Moreover, a domain may be represented by one or more directories. If multiple directories are deployed, user accounts and privileges need to be synchronized. Using the Resurrecting Duckling approach may be good for a person to manage his personal devices and services. With authentication and authorization aggregated on a device, it becomes easy for a person to keep a directory with him. Other types of control between services and directories may be used, for instance, servers may be used to authenticate and authorize users. Service access requests may occur via directories to the servers for authentication and authorization.

Directories may run on PCs, servers, or portable devices such as cell phones, PDAs, and Personal Servers [21]. These portable devices have reasonably good processing capability, network capability, and storage capacity. In our scenario, a directory may run on Bob's home PC to manage services in his house and another directory may run on his cell phone to manage devices that he carries, in his house, and/or in his office.

3.3 User Agent-Based Service Discovery

As discussed in the Introduction, it seems infeasible to require clients to support all potential authentication mechanisms in existing service discovery protocols. On the other hand, requiring users to remember services and their domains decreases usability. We use user agents to address the two problems. A user maintains all her identities in a table on her user agent. In the table, a domain identity is associated with a domain, an authentication method, authentication information (such as user name, password, or certificate, etc.), expiration time, and/or the domain's public key. Securing user agents is very important, but it is out of the scope of this paper.

Before a service discovery process, a client needs to bind to a user agent. This binding is short-term (one-time) and valid only in the current service discovery process. A secure channel may be established via side channels as discussed in [19], for example, physical touch between two devices becomes as a physical channel. By using these side channels, user agents and clients can exchange session keys and establish secure communication. Moreover, this binding also triggers a service discovery process.

In a service discovery process, a client queries a type of service and a user agent supplies necessary authentication information to gain privileges. The service query result depends on the identities that the user agent supplies. We will discuss our method in Section 4. Offloading authentication tasks from clients to user agents simplifies the design of clients, i.e., clients do not need to support various authentication mechanisms. Moreover, it is much easier to

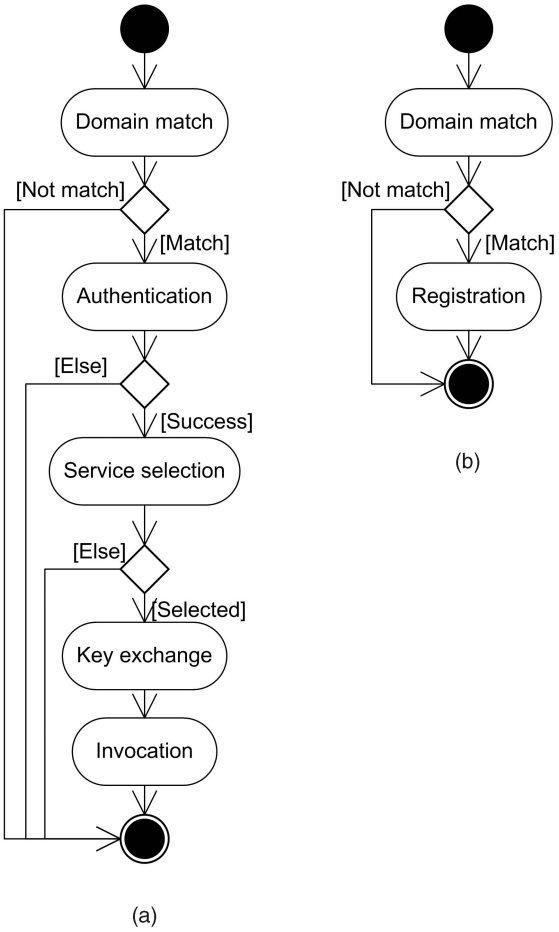

Fig. 2. The PrudentExposure activity diagrams. (a). The activity diagram of discovering a service. (b). The activity diagram of a service registration.

add a new authentication mechanism on a user agent than to add it on all clients.

Potential devices to serve as user agents should be handy and available whenever needed. Cell phones, PDAs, Personal Servers, and iButtons might be good candidates. iButtons [22] are very small and can be worn as a ring. One type of iButton is able to process various public key operations and is claimed to do a key operation within a second [22]. In our scenario, Bob and Alice may run their user agents on their cell phones, respectively. When Bob uses his Bluetooth earphone to discover MP3 songs, he uses the earphone to touch the cell phone. The touch binds his earphone to his user agent. His user agent supplies identities associated with his user roles and finds songs on his MP3 player and laptop. Similarly, if Alice uses Bob's earphone, the earphone binds to her user agent and, in turn, finds songs on her MP3 player and PDA. Alice's user agent should not find Bob's MP3 player unless Bob gives her an identity and grants her privileges.

4 The PrudentExposure Model

Before we focus on the interaction between a user agent and a directory, we briefly discuss our PrudentExposure model, as shown in Fig. 2. There are five major steps when discovering a service (Fig. 2a): domain match, authentication, service selection, key distribution, and invocation. First, a user agent searches available domains. After obtaining the information, it selects the correct identities to authenticate with them. Next, the user agent and the client ask the directories for service information. After receiving replies, the client or the user selects a service. Then, an encryption key is distributed to the selected service and the client. Last, the client is ready to use the service.

There are two steps in a service registration (Fig. 2b): domain match and registration. In the domain match step, a service discovers available domains. After finding its domain, a service sends an encrypted registration message using the secure channel, as we discussed in Section 3.2. The two domain match steps in Fig. 2a and 2b are very similar, thus, we will discuss only briefly the domain match step of the service registration in Section 4.7.

The domain match step is vital for user agents and directories. Without prudent exposure, the owner's and user's presence information and user identities may be sacrificed. To keep the domain match private and secure, user agents and directories speak *code words*. A user agent says a code word and then a directory checks whether or not the code word is correct. If the code word is correct, the directory says another code word and the user agent checks. This interaction establishes mutual trust between the user agent and the directory.

To address the issues of multiple coexisting domains and discovering at different places, a user agent may say many code words to find the existing domains. We express code words in Bloom filter form [4]. It allows user agents to say many codes words within one network packet. The advantages of using Bloom filters in our case are: security and privacy, simple code word assessments, space efficiency, and scalability.

4.1 An Introduction of Bloom Filters

Bloom filters are suggested as an efficient way to test membership [4]. The basic idea has two parts: Bloom filter generation and membership test. To generate a Bloom filter, as shown in Fig. 3a, select several hash functions that have the same range. The Bloom filter is represented as a bit array whose length equals to the range of the hash functions. Each possible hash result is represented as a bit in the Bloom filter. The filter is initially set to zero. Then, for a set of elements, apply every hash function to each element. By using a hash result as an index, we set a bit in the array. Note that a bit may be set many times due to different elements or due to different hash functions. For a membership test of an arbitrary element, the process is similar, as shown in Fig. 3b. First, apply every hash function to the element. Then, using each hash result as an index in the Bloom filter, any zero found at the index position in the filter means nonmembership. Otherwise, the element is considered as a member.

4.2 Matching Existing Domains Using Bloom Filters

An owner defines his domain, which is identified by a unique ID, as a *domain identity*. A domain identity is a secret that a domain shares with its users. Each code word exchanged between a user agent and a directory is the hash results of the domain identity. If three hash functions are used, then three bits of the hash results in the filter

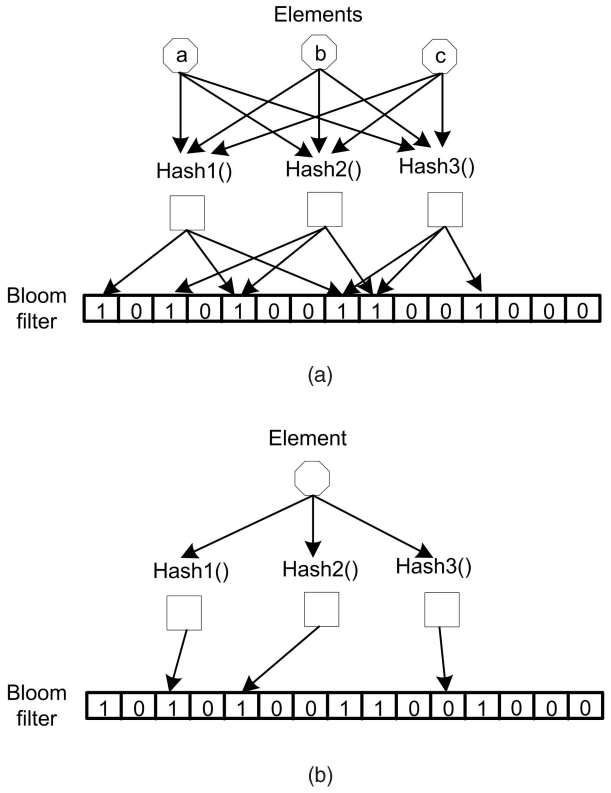

Fig. 3. A Bloom filter example using three hash functions. (a). Generating a Bloom filter. (b) A membership test.

represent a code word. To simplify the discussion, we suppose only one hash function is used to generate Bloom filters, thus, every code word is one bit.

To generate a bit in the filter, first, we calculate the hash of the domain identity using the function *h(domain identity)*. Then, we use the mod function, *mod(h(domain identity))*, to determine the bit to set in the filter. (Note that *mod(h(domain identity))* is equivalent to the hash function that we have discussed in Section 4.1, while *h(domain identity)* is an existing hash function that we utilize. Since we may think of the mod function as the last step of a Bloom filter hash function, we do not distinguish them later.) We choose MD5, SHA-1, and RIPEMD-160 as our hash functions because of their good properties of preimage resistance (computationally impossible to find the original message from the hash result) and collision resistance (computationally impossible to find two distinct messages with the same output) [23]. Therefore, given a bit in a Bloom filter, it is computationally difficult to know the original domain identity.

The domain match process in Bloom filter form is shown in Fig. 4. A user agent generates a Bloom filter by specifying necessary domain identities. Then, the Bloom filter is broadcast to directories for a membership test at the directories. If a directory finds a match, it generates another Bloom filter with two bits set. The first is the bit that the directory finds as a match and the second bit is set for the same domain identity using another hash function. Then, the filter is sent to the user agent for a membership test. It is not necessary to send the entire Bloom filter back to the client because there are only two bits set. A message that indicates which two bits are set is enough. Successfully

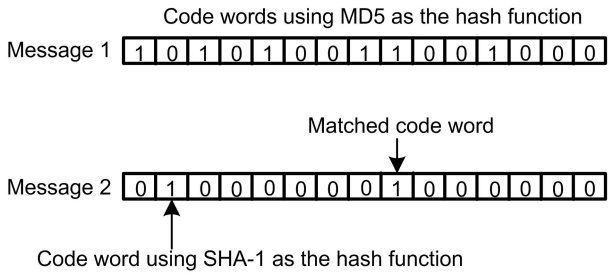

Fig. 4. Domain match using Bloom filters. A user agent broadcasts Message 1 and then a directory replies with Message 2 when it finds a matched code word.

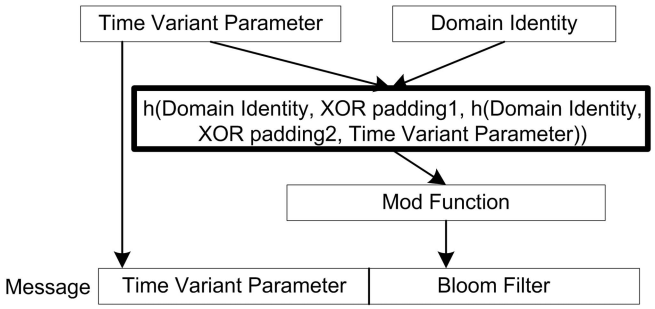

Fig. 5. Using dynamic information to generate Bloom filters.

generating the filters demonstrates the knowledge of the domain identity. User agents and directories agree on the hash functions in advance. User agents use MD5 to generate Bloom filters and directories use it to do membership tests. Similarly, directories use SHA-1 to generate reply messages and user agents use it to do membership tests. The RIPEMD-160 is used as an optional hash function for user agents to decrease the probability of false positive.

4.3 Exchanging Dynamic Code Words

Replay attacks are possible when exchanging static code words. In our case, an eavesdropper may associate bits with domains without finding the original domain identities. After listening to the domain match messages of a directory, he will find the bit in the Bloom filters associated with the domain by intersecting the filters. Next, he may replay the bit and then check the reply messages to test the existence of the domain. Furthermore, an eavesdropper may physically find out who the users of a domain are by checking the relative bits in the filters. Thus, static code words should be rarely used. However, they may be appropriate for some domains, such as commercial wireless services in airports, where a domain's presence information may not be important. Moreover, the computation overhead to generate dynamic code words is not a concern in most cases, as we show in Section 5.4.

To generate dynamic code words, we add a time variant parameter when calling the hash functions, specifically, hash-based message authentication codes as discussed in [23]. By adding this parameter, the code word serves as a one-time code word. Therefore, there is no fixed bit in the filters associated with a domain identity and a replay message can be easily detected.

The detailed method of setting a Bloom filter bit is highlighted in Fig. 5 and we use the hash algorithm

proposed in [24]. A domain identity is considered a key and the time variant parameter is considered a message. The time variant parameter includes a random number and a timestamp in our case. A user agent generates the time variant parameter and sends it along with the Bloom filter to directories. A directory uses the time variant parameter for the membership test and for the reply message. Moreover, a user agent uses the same time variant parameter for a discovery for all domains.

4.4 Preventing Internal Attackers to Act as an Owner

When a user agent finds a match in a reply message, it will try to authenticate. However, a domain identity is a shared secret and, thus, a reply message might come from an active internal attacker. If the message does come from an attacker, the attacker is rather sure when he sees the authentication message that a user from the domain is discovering services.

To prevent internal attackers, a directory may sign a reply message with its private key. At the user agent side, after a successful membership test, the user agent may verify the directory's signature using the directory's public key. Since an internal attacker only has the directory's public key, he is not able to sign the message. Thus, he is limited to analyze the Bloom filters as an internal eavesdropper can.

4.5 Obscure Identities to Improve User Privacy

Although an eavesdropper cannot act as an owner, he may still infer that a user is discovering services by analyzing the user's Bloom filter because every bit set in the filter represents a true code word. If the attacker has some knowledge of a user's code words, the user's presence information may be inferred. For instance, suppose Alice knows that Bob is discovering services and then knows that his discovery message has 18 bits set. If different employees in the office have a different number of bits set in their filters or if only a few employees set the same number of bits, then a filter with 18 bits set is very likely sent by Bob. Although a different group of 18 bits are set in each of Bob's discovery message (because of dynamic code words), the number of bits set may provide a clue that Bob is nearby.

To counter the attacks, a user agent may optionally set more bits in a Bloom filter to mix the true code words with other randomly selected bits. For instance, a user agent may set 80 bits in a Bloom filter, while he has 50 domain identities. Thus, an eavesdropper is unsure how many true code words a user has. To control the false positive rate at the directory side, our default is to set 1 percent of the bits in filters, including the true code words. Thus, query messages look the same.

If a user has many code words, we may use larger Bloom filters and compress them. For example, a user may send an 8K bytes Bloom filter with 650 bits set (1 percent). Such Bloom filters can be compressed to less than 1,000 bytes, as our simulation results show in Section 5.4.

4.6 Protecting Service Request Privacy

A service request specifies a service name and attributes. Nevertheless, it is not necessary to let all directories know the request. In our scenario, Bob may not want to tell the office's directory that he is looking for an MP3 player. Thus, instead of specifying the service name and attributes, a client may ask directories what services are available and authorized for it to access. This is similar to the wildcard search in many service discovery protocols [1]. Unlike those protocols, the reply messages are in the Bloom filter form in our model. The Bloom filters always fit in a single packet, while the message lengths of the existing approaches vary and may not be fit in one packet. Since services and attributes expressed in Bloom filter form were discussed in detail in SSDS [2], we do not further discuss them here. In short, Bloom filters can express many services and attributes in a filter. In SSDS, hierarchical directories need to handle many more services than in our model, thus, we are not concerned about the performance of building and rebuilding the Bloom filters. Unlike SSDS, queries in our model are evaluated at the client side instead of the directory side.

4.7 Putting It All Together—The Detailed Mechanism and Protocol

The message exchange sequence diagram of our model is shown in Fig. 6 and the protocol is shown in Fig. 7. The top halves in the figures are the service registration part, while the bottom halves are the discovering service part. In the first step of service registration (Step A in Fig. 6), a service utilizes the same approach that we discussed in Sections 4.3, 4.4, and 4.5: The service generates a Bloom filter using the domain identity and mixes the true code word with randomly selected bits. (An owner may use two different domain identities for users and services, respectively.) If a directory finds a match in Step B, it replies with another Bloom filter in Step C. Note that the directory sets two bits: One bit is that the directory finds itself as a match and the other bit is set by using another hash function as we have discussed in Section 4.2. When the service also finds a match, it sends a service registration message via the secure channel as we discussed in Section 3.2 (Steps D and E in Fig. 6). Afterward, the service periodically updates its service information. As long as updates succeed, the service does not send Bloom filters to discover directories. On the other hand, if the service fails to update service information or restarts, it may send out Bloom filters to discover directories.

In the first step of the discovering service part, a client sends a service request to a user agent using a session key shared between them as we described in Section 3.3. Then, in Step 2, the user agent sends a message asking for available domains. After receiving the request, a directory does a membership test to see if its domain matches the request in Step 3. If the directory finds a match in the Bloom filter, it creates a new Bloom filter and then sends the filter back in Step 4.

If there is a match in Step 5, the user agent authenticates with the directory. All messages afterward are encrypted using a session key exchanged between the user agent and the directory. The user agent forwards the service request in Step 6. Then, the directory matches the services and sends a reply message back in Steps 7 and 8. Next, the user agent forwards the messages to the client and lets the client select

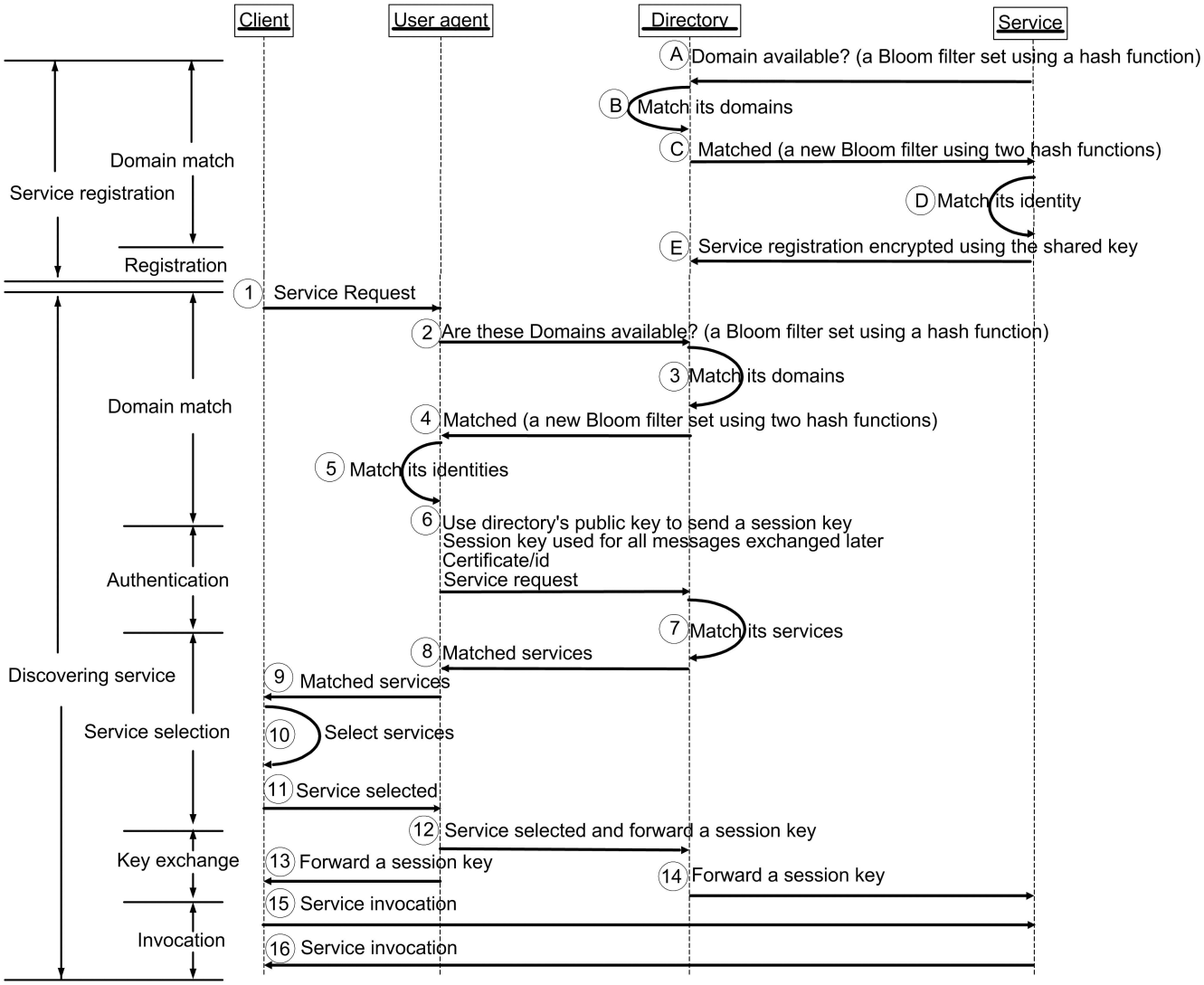

Fig. 6. The PrudentExposure message sequence diagram.

(Steps 9 and 10). After the client selects a service, it sends the request to the user agent and the user agent forwards it with another session key to the directory (Steps 11 and 12). This session key is for the client and the service to use. Next, the user agent and the directory forward the session key to the client and the service, respectively (Steps 13 and 14). Last, in Steps 15 and 16, the client and the service interact with each other.

5 System Evaluation

In this section, we first analyze the mathematical properties of our PrudentExposure model. Next, we consider the possible threats. Then, we formally verify our security protocol. Last, we discuss performance issues.

5.1 Mathematical Properties of the PrudentExposure Model

Bloom filter membership tests always recognize members correctly [4]. Thus, our method is complete such that matches in the original form are always matched in the Bloom filter form. However, the membership tests can have false positives, which mean nonmembers are recognized as members. When this happens, a waste of processing and communication occurs and privacy information may leak. If there is a false positive case at the directory side, a Bloom filter will be returned. If there are false positive cases at the user agent side, identities are supplied for authentication. In our case, the false positive rate is quite low. For example, if a filter is 8K bits long with 100 bits set, the false positive rate at the directory side is about 0.0124 given that the query is not sent by a domain user and, at the user agent side, it is less than 0.0001 given that the directory does not represent a domain that the user wants to discover. Increasing the length of Bloom filters, using more hash functions, or exchanging more rounds of Bloom filters can further decrease the false positive rate. More detailed discussion of the false positive rate in Bloom filters may be found in [18].

The mod function, *mod(8192)*, is a mapping function from 2^{128} space to 2^{13} space (MD5 is 128 bits, others are 160 bits). This means that 2^{115} possible hash results will set the same bit in a Bloom filter (supposing the hash result is evenly distributed in the 2^{128} space). It is not worthwhile for attackers to determine the hashes. Even if an attacker found

Notation:
C is a client; U is a user agent; D is a directory; S is a service.
M is a message.
t_X is a timestamp that X attaches.
R_X is a random number that X generates.
SR is a service request. MS is a matched service list.
PS is a service the client picks.
K_{XY} is a symmetric encryption key shared between X and Y.
UBF is a Bloom filter representing the domains that a user belongs.
SBF is a Bloom filter representing the domain that a service belongs.
DBF is a Bloom filter representing the domain a directory in charge of.
$()_{KX}^{-1}$ is X's signature using its signing private key.
$()_{KX}$ is an encryption using the encryption public key of X.
$()_{KXY}$ is an encryption using a symmetric key K shared between X and Y.

Msg	Sndr/Rcvr	Message	Step in Fig 6
a	S→D:	SBF, R_S, t_S	A
b	D→S:	DBF, R_S, t_S, (DBF, t_S) $)_{KD}^{-1}$	C
c	S→D:	(M, t_S)$_{KDS}$	E
1	C→U:	(SR, t_C)$_{KUC}$	1
2	U→D:	UBF, R_U, t_U	2
3	D→U:	DBF, R_U, t_U, (DBF, t_U) $)_{KD}^{-1}$	4
4	U→D:	(K_{UD}, t_U) $_{KD}$, (U, t_U, K_{UD}) $_{KU}^{-1}$, (SR, t_U)$_{KUD}$	6
5	D→U:	(MS, t_U, t_D)$_{KUD}$	8
6	U→C:	(MS, t_{U2}, t_C)$_{KUC}$	9
7	C→U:	(PS, t_{C2}, t_{U2})$_{KUC}$	11
8	U→D:	(PS, t_D, K_{CS})$_{KUD}$	12
9	D→S:	(K_{CS}, t_{D2})$_{KDS}$	13
10	U→C:	(K_{CS}, t_{C2})$_{KUC}$	14
11	C→S:	(M_1, t_{C3})$_{KCS}$	15
12	S→C:	(M_2, t_{C3})$_{KCS}$	16

Fig. 7. The PrudentExposure protocol.

a hash, it is still mathematically impossible to find the original domain identity from the hash.

5.2 Threats Analysis

Since user agents and directories have a priori knowledge, such as public/private keys or user names and passwords, they can set up session keys. Thus, all the messages they exchange can be encrypted, as shown in Fig. 6, Step 6 and afterward. Hence, we focus on how a user agent correctly identifies directories and how a directory correctly identifies a user.

Unlike the discussion in Section 5.1 that assumes both directories and user agents are honest, user agents and directories might be malicious. There could be external or internal attackers. To prevent internal attackers, we may use the technique discussed in Sections 4.4 and 4.5.

External attackers, on the other hand, will not gain much because Steps 2-5 in Fig. 6 are for a user agent to identify the existence of the directories. Acting as a user agent, the attacker may cause a directory to respond. In this case, the probability of a false positive using one hash function for the directory is

$$p(falsepositive \mid nonmember) = \frac{M}{L},$$

where M is the number of bits set in a Bloom filter and L is the filter length. Since a large number of bits set causes a higher false positive rate, a very high ratio of 1s set in a Bloom filter is suspicious. Directories check the number of bits set in a filter. If the number of the bits set is more than a threshold (5 percent), directories do not further check whether there is a match in the filter. By default, the filter length is 8,192 bits and 5 percent allows a user to set up to 410 code words. Similarly, an attacker that keeps sending Bloom filters with a majority of the bits set differently is also suspicious. Directories do not remember the states before authentication. Therefore the attacks will cause a limited waste of resources.

Acting as a directory, the chance of guessing the correct answer is

$$p(falsepositive \mid nonmember) = \frac{1}{L}.$$

Moreover, if the signature is required, as we discussed in Section 4.4, it is considered computationally difficult for an attacker to sign a reply message. Another possible attack is to replay reply messages heard from other directories. A replay message, however, is easy to detect since a user agent has already seen it from the genuine directory.

TABLE 1
Formal Verification Using BAN Logic

Step	Idealized Protocol	Stepwise results
1	$\{SR, \#(SR)\}_{KUC}$	$U \mid\equiv SR$
2	UBF	$D \mid\equiv UBF$
3	DBF, $\{DBF, t_U\}_{KD}^{-1}$	$U \mid\equiv DBF$
4	$\{U \overleftrightarrow{K_{UD}} D, \#(U \overleftrightarrow{K_{UD}} D)\}_{KD}$, $\{t_U, U, U \overleftrightarrow{K_{UD}} D\}_{KU}^{-1}$, $\{SR, \#(SR)\}_{KDU}$	$D \mid\equiv U \overleftrightarrow{K_{UD}} D$, $D \mid\equiv SR$
5	$\{MS, \#(MS)\}_{KUD}$	$U \mid\equiv MS$
6	$\{MS, \#(MS)\}_{KUC}$	$C \mid\equiv MS$
7	$\{PS, \#(PS)\}_{KUC}$	$U \mid\equiv PS$
8	$\{PS, \#(PS), C \overleftrightarrow{K_{CS}} S, \#(C \overleftrightarrow{K_{CS}} S)\}_{KUD}$	$D \mid\equiv PS$, $D \mid\equiv C \overleftrightarrow{K_{CS}} S$
9	$\{C \overleftrightarrow{K_{CS}} S, \#(C \overleftrightarrow{K_{CS}} S)\}_{KDS}$	$S \mid\equiv C \overleftrightarrow{K_{CS}} S$
10	$\{C \overleftrightarrow{K_{CS}} S, \#(C \overleftrightarrow{K_{CS}} S)\}_{KUC}$	$C \mid\equiv C \overleftrightarrow{K_{CS}} S$
11	$\{C \overleftrightarrow{K_{CS}} S, \#(C \overleftrightarrow{K_{CS}} S)\}_{KCS}$ from C	$S \mid\equiv C \mid\equiv C \overleftrightarrow{K_{CS}} S$
12	$\{C \overleftrightarrow{K_{CS}} S, \#(C \overleftrightarrow{K_{CS}} S)\}_{KCCS}$ from S	$C \mid\equiv S \mid\equiv C \overleftrightarrow{K_{CS}} S$

5.3 Formal Verification

We formally verify our protocol using BAN logic [5] and extend the logic to meet our needs. It assists us in improving our protocol. During a few rounds of the design and verification processes, the logic helps us find a subtle bug. The detailed notation may be found in [5]. As a convention of the verification process, we first convert the protocol to an idealized protocol. Next, we list all the assumptions. Then, we deduct step-by-step based on the logical postulates to reach our conclusion. The deduction is quite lengthy. We show the idealized protocol and stepwise results in Table 1. (The verification of the service registration part is very similar to Steps 2, 3, and 9 of the discovering service part, thus we omit it.) Step 1 is trivial. Since Step 4 to Step 11 is a procedure of authentication and key distribution, the use of BAN logic is straightforward. We show the most complicated step, Step 4, in Table 2, and omit the similar and lengthy discussion of the other steps.

TABLE 2
Verification of Step 4 in Table 1

Micro steps	Deduction	Notes
1	$\frac{D \triangleleft \{K_{UD}\}_{K_D}, D \mid\equiv \overset{K_D}{\rightarrow} D}{D \triangleleft K_{UD}}$	Using the other rule. The directory sees the session key that the user agent wants to share with it.
2	$\frac{D \triangleleft \{K_{UD}\}_{K^{-1}}, D \mid\equiv \overset{K_U}{\rightarrow} U}{D \mid\equiv U \mid\sim K_{UD}}$	Using the message-meaning rule. The directory believes that the user agent once said the session key.
3	$\frac{D \mid\equiv U \mid\sim K_{UD}, D \mid\equiv \#(K_{UD})}{D \mid\equiv U \mid\equiv K_{UD}}$	Using the nonce-verification rule. The directory believes that the session key is recent and the user agent still believes in it.
4	$\frac{D \mid\equiv U \mid\equiv K_{UD}, D \mid\equiv U \mid\Rightarrow K_{UD}}{D \mid\equiv K_{UD}}$	Using the jurisdiction rule. The directory believes the session key.
5	$\frac{D \mid\equiv U \overset{K_{UD}}{\leftrightarrow} D, D \triangleleft \{SR\}_{K_{UD}}}{D \mid\equiv U \mid\sim SR}$	Using the message-meaning rule. The directory believes the user agent once said the service request.
6	$\frac{D \mid\equiv U \mid\sim SR, D \mid\equiv \#(SR)}{D \mid\equiv U \mid\equiv SR}$	Using the nonce-verification rule. The directory believes that the service request is recent and the user agent still believes in it.
7	$\frac{D \mid\equiv U \mid\equiv SR, D \mid\equiv U \mid\Rightarrow SR}{D \mid\equiv SR}$	Using the jurisdiction rule. The directory believes the service request.

The micro Steps 4 and 7 in this table are shown in Table 1 as stepwise results.

Since Steps 2 and 3 are not authentication messages, the logic cannot be used directly. We extend the logic to help us check these two steps because the power of the BAN logic is its ability to check the freshness and binding. We extend the logic constructs as follows:

- $(\mathbf{M} \subset \mathbf{G})$: M is a member of group G who knows the shared secret.
- **P[BF]Y**: P finds a match in a Bloom filter which uses Y as a secret. This only means that there is a possibility that the Bloom filter generating party knows the secret Y. The probability is as we discussed in Section 5.1 and Section 5.2.
- $\mathbf{P} \overset{Y}{\infty} \mathbf{G}$: P shares a secret Y with a group G. In our case, G is the group of users of the domain P.

We also add the following postulates:

$$\frac{\mathrm{P} \overset{Y}{\infty} \mathrm{G},\ \mathrm{P[BF]_Y}}{\mathrm{P}|\equiv (\mathrm{M} \subset \mathrm{G})|\sim \mathrm{BF}}, \tag{1}$$

$$\frac{\mathrm{P}|\equiv (\mathrm{M} \subset \mathrm{G})|\sim \mathrm{BF},\ \#(\mathrm{BF})}{\mathrm{P}|\equiv (\mathrm{M} \subset \mathrm{G})|\equiv \mathrm{BF}}, \tag{2}$$

$$\frac{\mathrm{P}|\equiv (\mathrm{M} \subset \mathrm{G})|\equiv \mathrm{BF},\ (\mathrm{M} \subset \mathrm{G})|\Rightarrow \mathrm{BF}}{\mathrm{P}|\equiv \mathrm{BF}}. \tag{3}$$

The first postulate, (1), states that if P finds a match in a Bloom filter using secret Y, then there is a probability that one member M of group G generates the Bloom filter. Therefore, we are clear that if the Bloom filter is generated without a time variant parameter, the filter may be replayed. The second postulate, (2), goes further based on the freshness of the Bloom filter. It comes to the conclusion that P believes that one member of its user group generates the filter with a certain probability. Since the member has the control over the generation of the Bloom filter, P believes the Bloom filter (Postulate (3)). Based on these postulates, we can mechanically deduct and get the results for Steps 2 and 3, as shown in Table 1.

Moreover, the logic forces us to explicitly write down our assumptions to clarify our design goals. Our protocol is targeted for wired or wireless LAN environments. If used beyond LAN environments, given that a user agent and a directory cannot directly hear each other's broadcast or multicast messages, the messages may be replayed in real-time without notice. Furthermore, the time stamp and the random number used as the time variant parameter require accurate internal clocks in the user agents and directories. (The clocks do not drift hours away. Otherwise, user agents and directories have to use large caches to check the validity of the timestamps.)

5.4 Performance Discussion

Our model is based on the client-service-directory model. Compared to that model, our model is more efficient because a directory only replies when a match is found. However, three additional messages are needed in comparison to the insecure client-service-directory model. Two messages are for the user agent and the directory to send the session key to the client and the service, respectively. The other message is for the user agent to notify the directory the selected service and the session key. The three messages should not introduce much overhead.

Our concerns are the overheads that user agents and directories need to calculate Bloom filters and do membership tests (Steps 2-5 in Fig. 6), and that every party does symmetric key encryption and decryption operations (from Step 6 to Step 16).

TABLE 3
Performance Evaluations on Bloom Filter Exchanging between User Agents and Directories

Component	Task	Time	Step in Fig. 6
User agent	Generate a Bloom filter with 100 dynamic code words	15.62ms	2
User agent	Send a Bloom filter	2.52ms	2
Directory	Check whether there is a match in the Bloom filter	<1ms	3
Directory	Generate and send a Bloom filter	5.06ms	4
User agent	Waiting time after sending a filter until receiving a reply	11.62ms	
User agent	Check whether there is a match in the Bloom filter	<1ms	5

We measure our prototype system on Compaq iPAQs running Microsoft PocketPC 3.0. Each PDA has an ARM SA1110 206 MHz processor, 64MB RAM, an expansion pack, and a D-Link DCF-650W wireless card. The wireless cards are set to the 802.11 ad hoc mode with 2Mbps. Our software is developed using Microsoft eMbedded Visual C++ 3.0.

We measure the performance of user agents and directories exchanging Bloom filters (Steps 2-5 in Fig. 6). In our prototype, this part of the protocol uses UDP multicast. (Multicast or broadcast communication may not be reliable. More complex algorithms may be used to improve reliability such as the Multicast Convergence Algorithm in SLP [11].) User agents and directories are configured to use a multicast address for query messages and another multicast address for reply messages. Table 3 shows the average time of the tasks in 100 discovery processes. It takes a user agent about 20 milliseconds to generate a Bloom filter with 100 code words and to send a query message. A directory takes less than 1 millisecond to check whether there is a bit that matches its code word. If there is a match, a directory spends about 5 milliseconds to generate another Bloom filter and send back a reply message. Next, it takes less than 1 millisecond for a user agent to check whether the Bloom filter in the reply message matches the code word. In summary, it takes a user agent about 30 milliseconds from the time to generate a filter to finish processing the first reply message.

From Step 6 to Step 16 in Fig. 6, when compared to an insecure model, every step needs some symmetric key encryption and decryption operations. Our previous experience with respect to building a secure service discovery protocol shows that the overhead to do secure key operations are rather efficient on PDAs [25]. The public key operations take hundreds of milliseconds, while symmetric key operations and hash operations take a few to dozens of milliseconds. Thus, we believe that discovering a service should be possible within a reasonable time period.

The compression ratio of the Bloom filters that we use is high, as shown in Fig. 8. We randomly generate 1K, 2K, 4K, 8K, and 16K bytes Bloom filters with 5 percent, 1 percent, and 0.5 percent of the bits set in the filters. We generate 1,000 filters for each combination of the rate and length. Next, we use zlib (a data compression library) version 1.1.4 for Windows CE and select the maximum compression option to compress the filters. Last, we determine the largest file size for each combination of the rate and length. The compression takes from 20 milliseconds for 1K bytes filters up to 600 milliseconds for 16K bytes filters on the same PDAs as we described above. Thus, we may compress Bloom filters in the protocol without affecting the performance much.

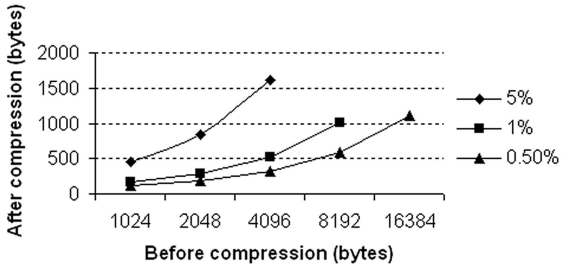

Fig. 8. Simulation results of the compression ratio of Bloom filters with different percentage of bits set.

6 Conclusion and Future Work

In this paper, we have proposed a private, secure, and efficient service discovery model. The PrudentExposure model provides an efficient way for authorized users to discover services while hiding services from unauthorized users. It automatically selects the right identities for service accesses. We protect sensitive information such as service information, user identities, user's presence information, domain identities, owner's presence information, and service requests. We have analyzed our model mathematically and have verified our protocol formally. Our prototype system shows that our method is efficient.

We are developing on mechanisms to monitor and control the accesses of the services and, thus, provide a method for users to easily view the status of the services and the history of service accesses. To coexist with the current models (providing services without a privacy concern), the security checks at the directories and services may be loosened. One possible solution is to reserve bits in the Bloom filters to represent domains that do not need identities for accesses. Meanwhile, the directories will not check user privileges and always reply with the availability of the services.

We are investigating revocation of domain identities for our model. Revocation of a domain identity from a user is a problem, when many users share the domain identity as a secret. A new domain identity needs to be distributed to all other users, but the users may not be online or in the vicinity. One possible solution is that a directory shares different secrets with different users. Thus, an owner can easily stop a user from discovering the existence of his domain. This approach increases the processing overhead and the false positive rate on directories and the processing overhead linearly depends on the number of users. If a domain has up to dozens of users, this solution is very efficient (up to dozens of milliseconds for a directory to verify a user). If a domain has hundreds or thousands of users, a directory may alternatively share different secrets with different groups of users. Revocation of a secret from a user involves distributing a new secret to the rest of the users in the group. Moreover, it is also feasible to use hashes of two identities as a code word, one to identity a group and one to identify a

user within a group. Distribution of a new shared group identity is not urgent after revocation, because only one combination of the two identities is a valid code word. Nevertheless, if each code word represents a user in a domain, all domains in the vicinity know the user is nearby when the user sends a Bloom filter.

We are also working to extend our model to discover services without the presence of directories. For example, how to authenticate and authorize Bob, if he wants to access Alice's electronic books but Alice and her directory are not in the office? Maintaining user names and passwords seems costly for limited resource devices themselves. Managing user names and passwords for every device is infeasible for both owners and users. Authorization certificates may simplify the process on the devices, but acquiring and maintaining the certificates for each device on the user's side introduces much administrative overhead. Thus, without support from online directories, user authentication, authorization, and privacy (as we discussed in the Introduction) for the independent services seem very challenging.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable comments that greatly helped the authors improve this paper. Their research was supported in part by US National Science Foundation Grant No. 0334035, US National Institutes of Health Grant No. EB002238-01, and Hong Kong RGC Grants HKUST6264/04E and AoE/E-01/99. This paper is an extension of a paper that appeared in the *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications,* March, 2004.

REFERENCES

[1] F. Zhu, M. Mutka, and L. Ni, "Service Discovery in Pervasive Computing Environments," *IEEE Pervasive Computing,* vol. 4, pp. 81-90, 2005.
[2] S. Czerwinski, B.Y. Zhao, T. Hodes, A. Joseph, and R. Katz, "An Architecture for a Secure Service Discovery Service," *Proc. Fifth Ann. Int'l Conf. Mobile Computing and Networks (MobiCom '99),* 1999.
[3] C. Ellison, "Home Network Security," *Intel Technology J.,* vol. 6, pp. 37-48, 2002.
[4] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Comm. ACM,* pp. 422-426, 1970.
[5] M. Burrows, M. Abadi, and R. Needham, "A Logic of Authentication," *ACM Trans. Computer Systems,* 1990.
[6] Sun Microsystems, "Jini Technology Core Platform Specification," Version 2.0, http://wwws.sun.com/software/jini/specs/, 2003.
[7] UPnP. Forum, "Universal Plug and Play Device Architecture 1.0," http://www.upnp.org/resources/documents/CleanUPnPDA101-20031202s.pdf, 2003.
[8] S. Cheshire and M. Krochmal, "DNS-Based ServiceDiscovery," Apple Computer, http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt, 2004.
[9] Bluetooth SIG, "Specification of the Bluetooth System," http://www.bluetooth.org/, 2004.
[10] Salutation Consortium, "Salutation Architecture Specification," ftp://ftp.salutation.org/salute/sa20e1a21.ps, 1999.
[11] E. Guttman, C. Perkins, J. Veizades, and M. Day, "Service Location Protocol, Version 2," http://www.ietf.org/rfc/rfc2608.txt, 1999.
[12] M. Nidd, "Service Discovery in DEAPspace," *IEEE Personal Comm.,* pp. 39-45, 2001.
[13] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The Design and Implementation of an Intentional Naming System," *Proc. 17th ACM Symp. Operating Systems Principles (SOSP '99),* 1999.
[14] M. Balazinska, H. Balakrishnan, and D. Karger, "INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery," *Proc. Pervasive 2002—Int'l Conf. Pervasive Computing,* 2002.
[15] P. Eronen and P. Nikander, "Decentralized Jini Security," *Proc. Network and Distributed System Security Symp. (NDSS 2001),* 2001.
[16] R. Rivest and B. Lampson, "SDSI—A Simple Distributed Security Infrastructure," http://theory.lcs.mit.edu/~rivest/sdsi10.html, 1996.
[17] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, "SPKI Certificate Theory," http://www.ietf.org/rfc/rfc2693.txt, 1999.
[18] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Trans. Networking,* vol. 8, pp. 281-293, 2000.
[19] F. Stajano and R. Anderson, "The Resurrecting Duckling: Security Issues for Ad-Hoc Wireless Networks," *Proc. Seventh Int'l Workshop Security Protocols,* 1999.
[20] F. Stajano and R. Anderson, "The Resurrecting Duckling—What Next?" *Proc. Eighth Int'l Workshop Security Protocols,* 2000.
[21] R. Want, T. Pering, G. Danneels, M. Kumar, M. Sundar, and J. Light, "The Personal Server—Changing the Way We Think about Ubiquitous Computing," *Proc. Fourth Int'l Conf. Ubiquitous Computing,* 2002.
[22] iButton Home Page, http://www.ibutton.com/, 2003.
[23] A. Menezes, P.v. Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography,* pp. 321-383. CRC Press, 1996.
[24] M. Bellare, R. Canettiy, and H. Krawczykz, "Keying Hash Functions for Message Authentication," *Proc. Advances in Cryptology (CRYPTO '96),* 1996.
[25] F. Zhu, M. Mutka, and L. Ni, "Splendor: A Secure, Private, and Location-Aware Service Discovery Protocol Supporting Mobile Services," *Proc. First IEEE Ann. Conf. Pervasive Computing and Comm.,* 2003.

Feng Zhu received the BS degree in computer science from East China Normal University in 1994 and the MS degree in computer science and engineering from Michigan State University in 2001. He is a PhD candidate in computer science and engineering at Michigan State University. He was a software engineer at Intel Co. from 1997 to 1999. His current research interests include pervasive computing, security for pervasive computing, computer networks, and distributed systems. He is a student member of the IEEE.

Matt W. Mutka received the BS degree in electrical engineering from the University of Missouri-Rolla in 1979, the MS degree in electrical engineering from Stanford University in 1980, and the PhD degree in computer science from the University of Wisconsin-Madison in 1988. In 1989, he joined the faculty of the Department of Computer Science, Michigan State University, East Lansing, Michigan, where he is currently an associate professor. He was a member of the technical staff at Bell Laboratories in Denver, Colorado, from 1979-1982, and a visiting scholar at the University of Helsinki, Helsinki, Finland, from 1988-1989 and in 2002. His current research interests include mobile computing, wireless networking, and multimedia networking. He is a senior member of the IEEE and a member of the IEEE Computer Society.

Lionel M. Ni received the PhD degree in electrical and computer engineering from Purdue University, West Lafayette, Indiana, in 1980. He is a professor and head of the Computer Science Department at the Hong Kong University of Science and Technology. A fellow of the IEEE and a member of the IEEE Computer Society, Dr. Ni has chaired many professional conferences and has received a number of awards for authoring outstanding papers. His research interests include parallel architectures, distributed systems, high-speed networks, and pervasive computing.