PrudentExposure: A Private and User-centric Service Discovery Protocol

Feng Zhu[1] Matt Mutka[1] Lionel Ni[2]

[1]*Dept. of Computer Science and Engineering*
Michigan State University
East Lansing, Michigan, USA
{zhufeng, mutka}@cse.msu.edu

[2]*Dept. of Computer Science*
Hong Kong University of Science and Technology
Kowloon, Hong Kong, China
{ni}@cs.ust.hk

***Abstract**: Service Discovery as an essential element in pervasive computing environments is widely accepted. Much active research on service discovery has been conducted, but privacy has been ignored and may be sacrificed. While it is essential that legitimate users should be able to discover services of which they have credentials, it is also necessary that services be hidden from illegitimate users. Since service information, service provider's information, service requests, and credentials to access services via service discovery protocols may be sensitive, we may want to keep them private. Existing service discovery protocols do not solve these problems. We present a user-centric model, called PrudentExposure, as the first approach designed for exposing minimal information privately, securely, and automatically for both service providers and users of service discovery protocols. We analyze the mathematical properties of our model and formally verify our security protocol.*

1. Introduction

As we are moving towards pervasive computing environments with billions of users, devices, and services, service discovery as an essential element is widely accepted. Many products and standards have emerged and much research work has been conducted. Privacy, however, has been ignored and therefore may be sacrificed when services may be discovered or used by any user. In this paper, we present a flexible, efficient, and scalable model, called *PrudentExposure*, to allow legitimate users to discover and use services easily, while it excludes others from discovering services' existence for wired and wireless LAN environments.

To help address the problems and solutions, we sketch a scenario of Bob discovering services at four places as shown in Figure 1. 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 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 does not want others to know what's in his bag. Nevertheless, he may wear his Bluetooth earphone and use it to discover his Bluetooth MP3 player and listen to songs on the MP3 player. In his office, Bob uses his computer, PDA, and MP3 player. When Bob goes to Alice's office, they look at a document on their office file server simultaneously with their respective laptops. 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 credentials for him to access. For example, his earphone should not discover Alice's Bluetooth MP3 player until Alice allows him to do so.

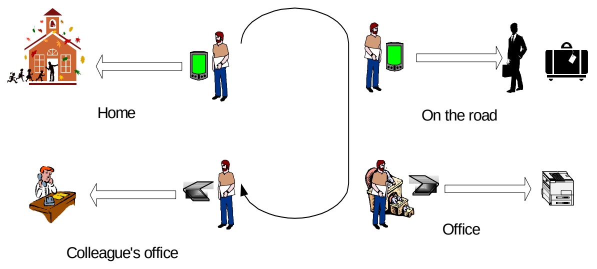

Figure 1. A service discovery scenario.

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 an owner of some services and devices. Most service discovery protocols do not consider security and privacy issues [1]. Devices and services are permitted to discover each other freely, so that security and privacy may be violated. Furthermore, exposing service information and service provider's information in an unrestricted manner is inclined to be attacked. Secure service discovery protocols, such as Secure Service Discovery Service (SSDS) [2], manage services centrally. Either these protocols may expose user and service privacy to a central directory or they may be appropriate for homogenous environments only. In heterogeneous environments, it is less likely that all devices and services are managed centrally because these devices and services may belong to different owners, such as Bob's MP3 player and Alice's MP3 player in our scenario.

Therefore, the user needs to use different credentials to access different services. As the number of credentials increases, manually selecting credentials during service discovery may be tedious. In addition, credentials may be sensitive and need to be kept private. UPnP Security [3] provides many authentication and authorization mechanisms, but the generic mechanism of automatically selecting services based on hashes is limited to one's own services. The approach is inefficient. Moreover, one common shortcoming of these existing protocols is that discovery is device-centric. Either the discovering device determines the discovery results, or it is assumed that the discovering devices have some ways to gain proper user credentials for certain services. It is highly risky if the discovering device determines the result, because it may totally violate access control of the services. For example, if Bob holds Alice's PDA, it will disclose all of Alice's personal services. Meanwhile, simply assuming that user credentials may be gained in some way is not appropriate. In order to provide security and privacy under this assumption, either complicated logic may need to be built in each discovering device or tedious user involvement may be necessary.

In PrudentExposure, explicit user information is required before a service discovery. Service management is owner-based. Discovery is user-centric and is based on the user roles. A user-centric model avoids the risk of the access violation in the device-centric model. Moreover, our PrudentExposure model manages user credentials centrally. Hence, the minimum processing requirements for discovering devices are reduced and the processing logic is simplified. We identify two basic goals for our model. First, privacy information including service information, service providers' identities and presence information, users' credential information, and service query information are exposing prudently. Second, an automatic scheme exposes only necessary credentials for the discovery processes. To our knowledge, there is no service discovery protocol that supports user and service privacy and meets the requirements as we discussed above. In PrudentExposure, privacy is protected before and during authentication, authorization, service selection, and invocation. Credential and service information is exchanged in a secure and private form, more specifically within Bloom filter form [4]. Further information exposure between a discovering party and a service provider is based on mutual matches within a step-by-step manner. Moreover, by protecting their sensitive information by hiding it and by not responding to arbitrary queries in the first place, devices and services may be less likely attacked.

We analyze the mathematical properties of our model and compare different design choices. Our approach is complete such that users do not miss any services that they should discover and a service lets all legitimate users discover it. Our privacy protocol is developed in a secure service discovery context. It is flexible to support various credentials and scalable to support many credentials. Moreover, we formally and mechanically verify our protocol by using and extending BAN logic [5]. 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 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 Universal Plug and Play (UPnP) [7], and Apple Computer's Rendezvous [8]. Service discovery protocols such as Bluetooth Service Discovery Protocol [9], Salutation [10], and Service Location Protocol Version 2 [11] are from different standardization organizations. Some representative academic projects are DEAPspace [12], Intentional Naming System (INS) [13], and INS/Twine [14]. Detailed comparison of these projects may be found in [1]. These protocols may be roughly classified as two models: *client-service model* and *client-service-directory model*. In simple environments such as home environments, the client-service model may be used. Clients first inquire about the services' availability. Comparing with the client's queries, matched services return replies. After receiving responses from services, clients select and contact services. To support thousands of computing services, such as the services in enterprise environments, directories may be used to store the service information. In the client-service-directory model, a client queries a directory for service information and then contacts services. Services, on the other hand, register service information with directories. In these two service discovery models, user information is not considered. Therefore, any services may be discovered and used by any user. Otherwise, it is assumed that there are some ways to gain user information. This assumption is not appropriate in pervasive computing environments, especially considering the heterogeneous capability of the devices that need to handle many different roles of users.

As one of the first secure service discovery protocol, SSDS from UC Berkeley has many built-in security features [2]. In SSDS, directories, known as Service Discovery Service Servers, are in a hierarchical structure and all directories are trusted. Clients and services authenticate with the directories for service lookups and announcements, respectively. Various other security features are considered including authorization, data and service privacy, and integrity. In short, SSDS is good for environments, such as enterprise environments, where clients are willing to expose identities and service requests, and where services are willing to expose their service information to central directories. In pervasive environments, however, central directories may not be appropriate. For example, as a service owner, Bob may not be willing to register his MP3 player with the office's directory. On the other hand, the office's directory may not accept a request that Alice wants to register her MP3 player. The situation of both Bob's directory and the office's directory coexisting is not addressed. The consequence might be that Bob has to supply credentials manually. Thus, SSDS may not work well within pervasive environments and does not meet our requirements as discussed in the Introduction section.

UPnP Security [3] offers many security solutions for environments such as home environments. 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 has 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 owning the device or not. One shortcoming of the approach is that all devices need to respond back to a client. 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 back to the service discovery request. Although Bob's earphone will tell which MP3 player is Bob's, the process is not efficient. Among all services that reply, users may not have privileges to access many of them and should not even be aware of the existence of the services.

Other work also influences our approach, such as SDSI [15] and SPKI [16]. We borrow the ideas of using public keys and hashes of public keys to represent principals (e.g., users and organizations). We also use Bloom filters [4] extensively in our protocol because using Bloom filters provides a good method for membership tests. In Summary Cache [17], 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 Bloom filter form when building the hierarchical directory structures. The Resurrecting Duckling security policy [18, 19] provided 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 the idea to associate owners' services with their directories.

3. Service Discovery Architecture

In our service discovery architecture, there are four types of components – clients, services, directories, and user agents. The architecture is based on the client-service-directory model [1]. Two discovery diagrams are shown in Figure 2. Clients and services are similar to their counterparts in the client-service-directory model. Directories, however, are different from the existing protocols. User agents manage credentials for users. We discuss directories and user agents first. Then, we discuss two bindings in bootstrapping as steps A and B in Figure 2. The detailed discovery process, as unlabelled arrows in Figure 2, is discussed in Section 4.

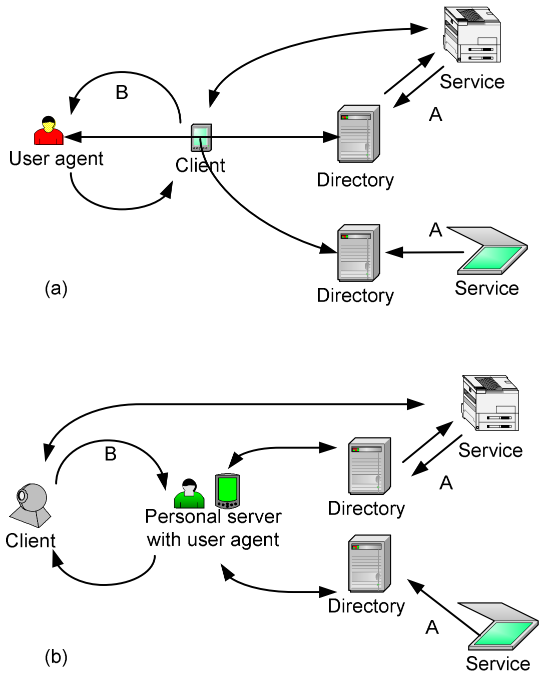

Figure 2. Two service discovery models.

3.1. Owner-based Service Management

In our model, an owner defines its domain represented by a public encryption key and assigns privileges to its users. Directories store information of the devices and services that belong to their owners. It may also store information of devices or services, which are temporarily owned by the directory's owner. For example, store rental car information in one's directory.

An owner may have a few domains, for instance, a long-term domain and a temporary domain. The long-term domain of a user may be used to share services with family members. At the same time, the user may have a temporary public key to represent a temporary domain to share services with another user instead of creating a certificate or a temporary user in the long-term domain. Moreover, by moving the requirements onto directories, owner-based service management may reduce the minimum security and privacy requirements on services.

3.2. User Agent-based Service Discovery

Users may have many digital credentials to represent their roles in real life. For example, Bob may have credentials from work, from other family members, from friends, or for anonymous uses. We think that the idea proposed in SDSI/SPKI that each user is a certification authority is attractive for pervasive computing environments. Thus, a user may have many credentials. The user agent manages credentials centrally for the user.

Before each discovery, a client needs to bind to a user agent. The user agent supplies all credentials that are needed for service discovery. In other words, when the same device is used by two users, the discovery results may be different because the user agents and therefore user roles are different. Moreover, user agent-based service discovery also simplifies the requirements for clients to authenticate users.

Various possible configuration approaches may exist depending on the devices on which the clients and user agents run. Figure 2 (a) shows a PDA discovering services, while Figure 2 (b) shows a digital camera without a public key operation capability discovering services via a user agent on a Personal Server [20] (a powerful processing and networking handheld device).

3.3. Bootstrapping

Before a service discovery starts, two bindings are necessary: a client needs to find a user agent and services need to announce information to directories. The binding between clients and user agents may be established via side channels. Then using these side channels, user agents and clients may share session keys. Detailed discussion may be found in [21]. The binding between a service and a directory is long-term. We borrow the master-slave relationship from the Resurrecting Duckling approach [18, 19], a directory as a mother duck (master) and services as ducklings (slaves). After establishing the master-slave relationship, a service and a directory use a secure channel for communication. Service information expressed in directories, however, is short-term and fresh. Directories keep soft state of the service information. In other words, service information is announced with a life span. A service needs to announce its information again before expiration. Meanwhile, directories periodically flush out stale services. We also assume that before a service access, a service owner has assigned privileges to users.

4. The PrudentExposure Model

In our model, directories provide most security and privacy functionalities at the service provider side (services and directories). Likewise, at the discovering side (clients and user agents), user agents handle user credentials and most security and privacy information. Therefore, the problem is to solve the privacy and security issues between user agents and directories. If users have valid credentials from service providers, they can smoothly authenticate with the directories. Thus, the key problem is that user agents need to find the existing directories. During this discovering process, the user agents need to send out credential information and the domains need to release the domain information for these two parties to build up trust relationship with each other.

To keep this discovering process private and secure, user agents and directories may 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. Since there may be many directories in the vicinity, a user agent may say many code words in one message. We use a Bloom filter to represent directories that a user wants to contact. The advantages of using Bloom filters in our case are: security and privacy, simple code word assessments, space efficiency, and scalability. Eavesdroppers, however, may replay the code words, thus we use one-time code word to improve privacy for user agents and directories.

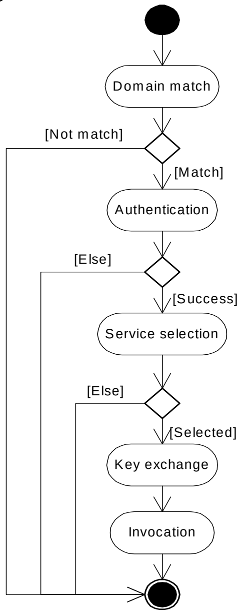

Figure 3. The PrudentExposure activity diagram.

The PrudentExposure model has five major steps, as shown in Figure 3, in the following order: domain match, authentication, service selection, key distribution, and invocation. First, a user agent looks for the domains that are available. After knowing the available domains, it selects the correct credentials to authenticate with them. Next, the user agent and the client ask the directories for service information. After receiving service information from directories, the client selects one. Then, an encryption key is distributed to the selected service and the client. Finally, the client uses the service. To limit the information exposure, the discovery process stops, if either the discovering side or the service provider side detects that it is impossible for a service access.

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 Figure 4 (a), select several hash functions first. The ranges of the hash functions are the same. 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. Given a set of elements, apply every hash function to each element. Then 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 Figure 4 (b). Apply every hash function to the element. Then use each hash result as an index to look up in the Bloom filter; any zero found at the index position in the filter means non-membership. The Bloom filters need to be rebuilt or updated when the set of members change.

Membership tests always recognize members correctly, but there may be false positive cases. False positive cases mean that non-members are identified as members. By adjusting the number of hash functions and the length of a Bloom filter, different false positive rates will occur. The possibility of false positives in our case is very low and thus the waste of calculation and communication is very limited as we will discuss in Section 5.1.

4.2. Protecting Credential Privacy

We use Bloom filters to represent domain identities. Each domain identity is represented as a bit in the filter and a bit is a code word. Many code words may be expressed in a Bloom filter. To generate the Bloom 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 find out the bit to set in the Bloom filter.[1] There are two Bloom filters used: one generated at the user agent side and the other generated at the directory side. A user agent generates a Bloom filter by specifying necessary credentials. Then the Bloom filter is sent to directories for a membership test. Likewise, if a directory finds a match, it will generate a Bloom filter indicating the domain that it manages and sends the Bloom filter to the user agent for a membership test. The Bloom filters may be calculated beforehand. Nevertheless, the filters need to be rebuilt, if a user has greater or fewer credentials or if a domain's identity changes.

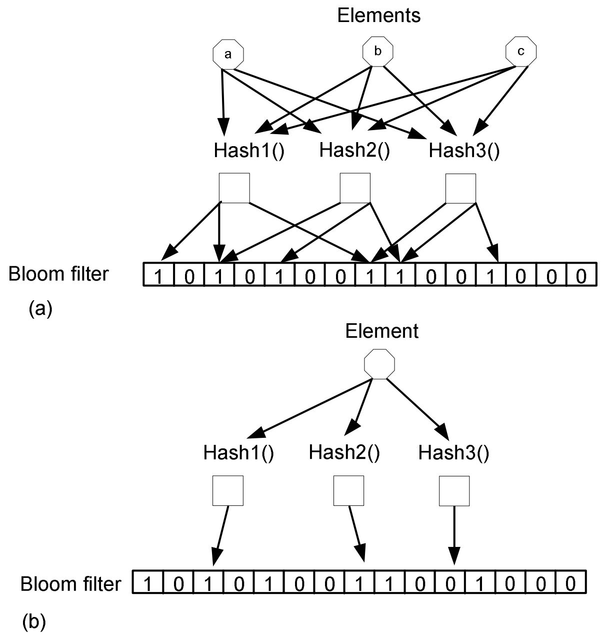

Figure 4. A Bloom filter example using 3 hash functions. (a). Generating a Bloom filter. (b) A membership test.

User agents and directories agree on hash functions in advance. For example, a user agent uses hash function, hash1(), to create a Bloom filter, and then a directory uses the same hash1() to do a membership test. Likewise, if there is a match, the directory uses hash function, hash2(), to generate the other Bloom filter and the user agent does a membership test using the same hash function. In this way, both the user agent and the directory know the other side has the knowledge of the domain identity. The method of using hash functions could be more flexible by specifying the functions in the messages.

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) [22]. By default, all user agents use MD5 as the hash function. The RIPEMD-160 is used as an optional hash function for user agents to decrease the probability of false positive (more detail in Section 5.1). All directories use SHA-1 as the hash function for the reply messages. The default Bloom filter length in the implementation is 8,192 bits

[1] Note *mod(h(domain identity))* is equivalent to the hash function that we discuss in Section 4.1, while *h(domain identity)* is an existing hash function that we utilize. Since we may think the mod function as the last step of a Bloom filter hash function, we do not distinguish them later.

(1024 bytes). After we calculate the hash value, we calculate the Bloom filter bit by *hash mod 8192.* Furthermore, compression operations may be used to decrease the length of the Bloom filters.

4.3. Exchanging Dynamic Information to Improve Privacy

One subtle problem of using static code words, bits in Bloom filters in our case, to represent domain identities is that an eavesdropper may physically associate bits with domains without finding the original identities. The eavesdropper may replay those bits and check the reply messages to test the existence of the directories. For some domains, such as public computing service in an office, privacy may not be important. For personal domains that represent users, better privacy may be necessary. Furthermore, an eavesdropper may physically find out who are the users of a domain by checking the relative bits in Bloom filters. Therefore he may figure out the relative credentials that the users have.

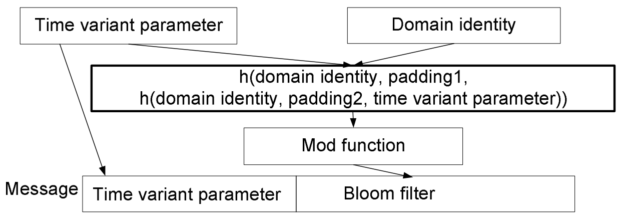

Figure 5. Using dynamic information to generate Bloom filters.

In order to hide the presence of domains and protect user credentials, we add a time variant parameter when calling the hash functions, specifically hash-based message authentication codes as discussed in [22]. The detailed method of setting a Bloom filter bit is highlighted in Figure 5. By adding a time variant parameter, a different bit may be set in the Bloom filter each time and a replay message can be easily detected. As shown in Figure 5, a domain identity is considered a key and the time variant parameter is considered a message. (To simplify explanation, we use a domain identity as a secret. Otherwise, a secret associated with the domain identity should be used.) The time variant parameter includes a random number and a timestamp in our case. A user agent generates the parameter and sends it along with the Bloom filter to directories. At the directory side, the same calculation will be performed. A match means a possibility that a user has the knowledge of the credential. When a domain initially creates a user, it notifies the user agent to calculate dynamic Bloom filter bits before each discovery. Moreover, a user agent uses the same time variant parameter for a discovery for all domains that require dynamic Bloom filters settings. Results of both dynamic and static (keyed and unkeyed) hash functions are set in the same Bloom filter in a message.

4.4. 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 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 Bloom filter form in our model. The Bloom filters always fits in a single message, while the message lengths of the existing approaches may vary, especially too many services matched to fit within a message. Since services and attributes expressed in Bloom filter form were discussed in detail in SSDS [2], we do not further discuss it here. In short, Bloom filters can express many services and attributes in a filter. In SSDS, hierarchical directories need to handle much 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.

4.5. The Detailed Mechanism and Protocol

The message exchange sequence diagram of our model is shown in Figure 6 and the protocol is shown in Figure 7. In the first step, a client sends a service request to a user agent using a session key shared between them. The shared key is generated during the bootstrapping process that we described in Section 3.3. Then in step 2, the user agent sends a message asking available domains. After receiving the request, a directory does a membership test to see if any of its domains 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 Bloom filter back in step 4. Note 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.

If there is a match in step 5, the user agent authenticates with the directory. All messages afterwards 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 (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.

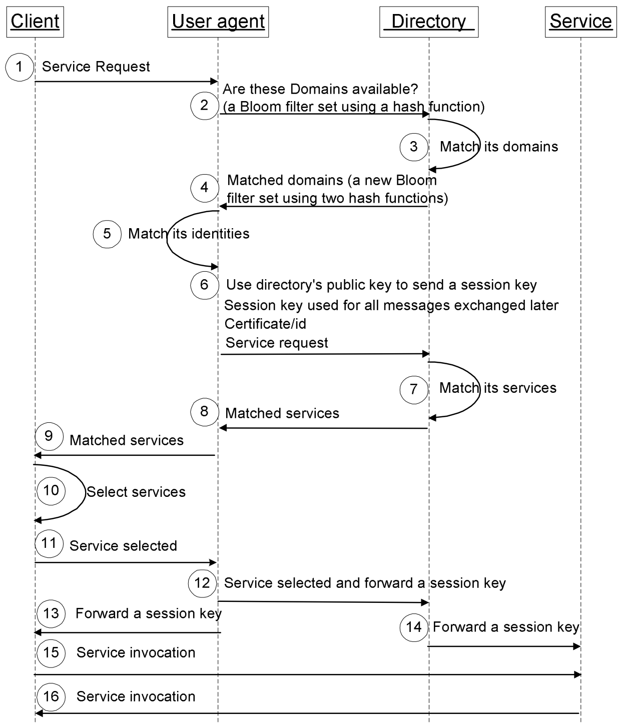

Figure 6. The PrudentExposure message sequence diagram.

5. System Evaluation

In this section, we first analyze the mathematical properties of our PrudentExposure model. Then, we consider possible attacks. Last, we formally verify our security protocol.

5.1. Mathematical Properties of the PrudentExposure Model

Bloom filters do not have false negative cases, as discussed in [4]. In other words, if a user has a credential for a domain, the relative bits in the filter will always be set correctly. Similarly, if a directory is the directory a user wants to contact, it will always set bits correctly in the reply messages. Therefore, our concern is the false positive cases. Given a directory is not a directory that a user wants to contact, what is the probability that we identify that the domain is one that the user wants to contact? The consequence of this false classification is that a user agent sends its encrypted credential to the directory. The directory, however, is not able to decrypt and thus the communication stops. Alternatively, given a user does not have a credential for

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. $(\ )_{KX^{-1}}$ is X's signature using its signing private key. $(\ )_{KX}$ is an encryption using the public encryption key of X.
$(\ )_{KXY}$ is an encryption using a symmetric key K shared between X and Y. UBF is a Bloom filter representing the domains that a user belongs. DBF is a Bloom filter representing the domain a directory in charge of.

Step	Sndr/Rcvr	Message	In Fig. 6
1	$C\rightarrow U$:	$(SR, t_C)_{KUC}$	1
2	$U\rightarrow D$:	UBF, R_U, t_U	2
3	$D\rightarrow U$:	DBF, R_U, t_U	4
4	$U\rightarrow D$:	$(K_{UD}, t_U)_{KD}$, $(U, t_U, K_{UD})_{KU^{-1}}$, $(SR, t_U)_{KUD}$	6
5	$D\rightarrow U$:	$(MS, t_U, t_D)_{KUD}$	8
6	$U\rightarrow C$:	$(MS, t_{U2}, t_C)_{KUC}$	9
7	$C\rightarrow U$:	$(PS, t_{C2}, t_{U2})_{KUC}$	11
8	$U\rightarrow D$:	$(PS, t_D, K_{CS})_{KUD}$	12
9	$D\rightarrow S$:	$(K_{CS}, t_{D2})_{KDS}$	13
10	$U\rightarrow C$:	$(K_{CS}, t_{C2})_{KUC}$	14
11	$C\rightarrow S$:	$(M_1, t_{C3})_{KCS}$	15
12	$S\rightarrow C$:	$(M_2, t_{C3})_{KCS}$	16

Figure 7. The PrudentExposure protocol.

a domain, what is the probability that we identify that the user has a credential? The consequence is that a directory participates in one round of wasteful computation and communication.

Now we look at the false positive cases quantitatively. In this subsection, the discussion of the probability of false positive is based on both user agents and directories are honest. (We will discuss malicious users and directories in Section 5.2.) A user agent calculates its credentials using one hash function and sets the relative bits in a Bloom filter. Given the Bloom filter with L bits and a user has C credentials, a bit will be set to one with probability:

$$p(match \mid nonmember) = \left[1-\left(1-\frac{1}{L}\right)^{C}\right] \qquad (1)$$

where $\left(1-\frac{1}{L}\right)$ is the probability that a bit is zero after one credential is calculated and inserted, and $\left(1-\frac{1}{L}\right)^{C}$ is the probability that a bit is zero after all necessary credentials are calculated and inserted. The false positive probability for each domain of a directory is exactly *(1)*. In Figure 8 (a), we show the probability of false positive using one hash function on various numbers of credentials. The Bloom filter length in bits is less than the maximum length of an Ethernet packet (12,000 bits). One way to reduce the probability of false positive is to use more than one hash function. Figure 8 (b) shows the probability of false positive using two hash functions, which is: $p(match \mid nonmember) = \left[1 - \left(1 - \frac{1}{L}\right)^{2C}\right]^2$.

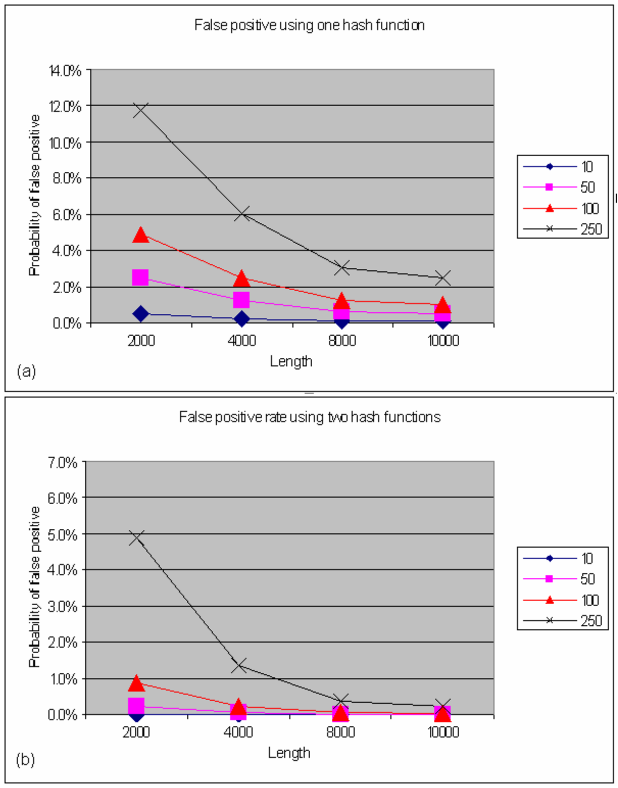

Figure 8. The probability of false positive of the PrudentExposure model at the directory side. (a). Using one hash function. (b). Using two hash functions

After a directory finds a match, it returns a Bloom filter, which has up to two bits set. The first bit is the bit that the directory finds as a match, and the second bit is set by using another hash function. The probability of false positive of each credential for a user agent is $p(match \mid nonmember) = (1) \times \frac{1}{L}$, which is very low. For example, for a 2K bits Bloom filter with 250 credentials encoded, the false positive rate is about 0.0001. Thus, a user is quite confident when there is a match. 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. If an owner has multiple domains that match a user's request, the bits for each domain are indicated in order in a message. Furthermore, including one more round of message exchange will dramatically decrease the probability of false positive. The probabilities of false positive at the directory side and the user agent side are $p(match \mid nonmember) = (1) \times \frac{1}{L} \times \frac{1}{L}$ and $p(match \mid nonmember) = (1) \times \frac{1}{L} \times \frac{1}{L} \times \frac{1}{L}$, respectively. More than one round of message exchanging is unnecessary in most cases.

The mod function, which we use is *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 there are 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 find out the hash. Even if an attacker found a hash, it is still mathematically impossible to find the original domain identity from the hash.

5.2. Threats Analysis

Since a user agent knows the public keys of the directories with which it wants to interact, all messages it exchanges with the directories can be encrypted, as shown in Figure 6 step 6 and afterwards. Thus, 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.

Internal attackers know the credentials, so they can always set the right bits in the Bloom filters regardless of whether static or dynamic information are exchanged. A user agent will believe an attacker as a directory. The attacker, however, is not able to decrypt the message, which is encrypted using the directory's public key. Thus, no security and privacy information leaks further.

External attackers on the other hand will not gain much because the steps 2-5 in Figure 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 false positive using one hash function for the directory is $p(falsepositive \mid nonmember) = \frac{M}{L}$, M is the number of bits set in a Bloom filter. Since the more bits set the higher the false positive rate, a very high ratio of 1's set in a Bloom filter is suspicious. 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 have a limited waste of resources. Acting as a directory, the chance of guessing the correct answer is $p(falsepositive \mid nonmember) = \frac{1}{L}$. Another possible attack is to replay reply messages heard from other directories. Even though a user agent is fooled and sends back an authentication message as the step 6 in Figure 6, the attacker is still not able to decrypt the message.

5.3. Formal Verification

We formally verify our protocol using BAN logic [5] and extend the logic to meet our needs. It assists us to improve our protocol. During a few rounds of design and verification processes, the logic helps to find a subtle bug and helps us to make decisions. The detailed notation may be found in [5]. As the convention of the verification process, we first convert the protocol to an idealized protocol. Next, we list all assumptions. Then, we deduct step-by-step based on logical postulates to reach our conclusion. The deduction is quite lengthy and we omit it. We only show the idealized protocol and stepwise results in Table 1 and omit the detailed discussion of the process. Step 1 is trivial. From Step 4 to Step 11 is a procedure of authentication and key distribution, thus we use the BAN logic smoothly.

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	$U \mid\equiv DBF$
4	$\{U \underset{\longleftrightarrow}{K_{UD}} D, \#(U \underset{\longleftrightarrow}{K_{UD}} D)\}_{KD}, \{t_U, U, U \underset{\longleftrightarrow}{K_{UD}} D\}_{KU^{-1}}, \{SR, \#(SR)\}_{KCU}$	$D \mid\equiv U \underset{\longleftrightarrow}{K_{UD}} D$, $D \mid\equiv SR$
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	$\{PS, \#(PS), C \underset{\longleftrightarrow}{K_{CS}} S, \#(C \underset{\longleftrightarrow}{K_{CS}} S)\}_{KUD}$	$D \mid\equiv PS$, $D \mid\equiv C \underset{\longleftrightarrow}{K_{CS}} S$
9	$\{C \underset{\longleftrightarrow}{K_{CS}} S, \#(C \underset{\longleftrightarrow}{K_{CS}} S)\}_{KDS}$	$S \mid\equiv C \underset{\longleftrightarrow}{K_{CS}} S$
10	$\{C \underset{\longleftrightarrow}{K_{CS}} S, \#(C \underset{\longleftrightarrow}{K_{CS}} S)\}_{KUC}$	$C \mid\equiv C \underset{\longleftrightarrow}{K_{CS}} S$
11	$\{C \underset{\longleftrightarrow}{K_{CS}} S, \#(C \underset{\longleftrightarrow}{K_{CS}} S)\}_{KCS}$ from C	$S \mid\equiv C \mid\equiv C \underset{\longleftrightarrow}{K_{CS}} S$
12	$\{C \underset{\longleftrightarrow}{K_{CS}} S, \#(C \underset{\longleftrightarrow}{K_{CS}} S)\}_{KCCS}$ from S	$C \mid\equiv S \mid\equiv C \underset{\longleftrightarrow}{K_{CS}} S$

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:

- **$(M \subset G)$:** 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.
- **$P \overset{Y}{\infty} G$:** 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:

$$\frac{P \overset{Y}{\infty} G,\ P[BF]_Y}{P \mid\equiv (M \subset G) \mid\sim BF} \quad (1) \qquad \frac{P \mid\equiv (M \subset G) \mid\sim BF,\ \#(BF)}{P \mid\equiv (M \subset G) \mid\equiv BF} \quad (2) \qquad \frac{P \mid\equiv (M \subset G) \mid\equiv BF,\ (M \subset G) \mid\Rightarrow BF}{P \mid\equiv BF} \quad (3)$$

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 the principal P believes that one member of its user group generates the filter with certain probability. Since that 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 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 multicast or broadcast 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 (not drift days) in the user agents and directories. We do not require synchronized clocks, but accurate clocks are necessary to make time stamps valid.

5.4. Performance Discussion

Our model is based on the client-service-directory model. Compared to that model, our model is more efficient in some aspects and has overheads in other aspects. Our model is more efficient that instead every directory replies, a directory only replies when a match is found. However, there are three additional messages are needed in comparison to the unsecured 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 that a service is selected and to forward the session key. Other overheads are that user agents and directories need to calculate Bloom filters and do membership tests (steps 2-5 in Figure 6); user agents authenticate with directories and do public key encryption and decryption operations (steps 5-6); and every party does some symmetric key encryption and decryption operations (from step 6 to step 16).

Our previous experience with respect to building a secure service discovery protocol shows that the overhead of doing secure key operations are rather efficient [23]. The public key operations take hundreds of milliseconds on PCs and PDAs, while symmetric key or hash operations are at dozens of milliseconds.

6. Conclusion

In this paper, we have proposed a private and user centric service discovery model. The PrudentExposure model provides an efficient way for authorized users to discover services simply, while hiding services from unauthorized users. It automatically selects the right credentials for service accesses. We protect privacy for service information, user credentials, domain identities, domain existence information, and service requests. We have analyzed our model mathematically and have proven the correctness of our protocol formally.

Potential devices as user agents might be Personal Servers [20] or iButtons [24] because these devices are handy and may be available whenever needed. Since Personal Servers' processing capability, network capability, and storage capacity are as good as laptops. They may serve as proxies or user agents, via which less processing power devices discover services. iButtons on the other hand are very small and can be worn as a ring. One type of the iButtons is able to process various public key operations within a second [24]. We are also working on extending our model to discover services without presence of directories.

Acknowledgement

This research is supported in part by NSF Grants No. 0334035, 0098017, 9911074, NIH Grant No. EB002238-01, a grant from Microsoft Research, and Hong Kong RGC Grants HKUST6161/03E and DAG02/03.EG02. The authors thank anonymous reviewers for their valuable comments.

References

[1] F. Zhu, M. Mutka, and L. Ni, "Classification of Service Discovery in Pervasive Computing Environments," Michigan State University, East Lansing, available at http://www.cse.msu.edu/~zhufeng/ServiceDiscoverySurvey.pdf MSU-CSE-02-24.
[2] S. Czerwinski, B. Y. Zhao, T. Hodes, A. Joseph, and R. Katz, "An Architecture for a Secure Service Discovery Service," presented at Fifth Annual International Conference on Mobile Computing and Networks (MobiCom '99), Seattle, WA, 1999.
[3] C. Ellison, "Home Network Security," Intel Technology Journal, vol. 06, pp. 37-48, 2002.
[4] B. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," Communications of ACM, pp. 422-426, 1970.
[5] M. Burrows, M. Abadi, and R. Needham, "A Logic of Authentication," ACM Transactions on Computer Systems, 1990.
[6] Sun Microsystems, "Jini™ Architecture Specification," Sun Microsystem, available at http://wwws.sun.com/software/jini/specs/ December, 2001.
[7] Microsoft Corporation, "Universal Plug and Play Device Architecture," Version 1.0, Microsoft Co., 2000, available at http://www.upnp.org/download/UPnPDA10_20000613.htm.
[8] S. Cheshire, "Discovering Named Instances of Abstract Services using DNS," Apple Computer, 2002, available at http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt.
[9] Bluetooth SIG, "Specification of the Bluetooth System -- Core," available at http://www.bluetooth.org/docs/Bluetooth_V11_Core_22Feb01.pdf, February 22, 2001.
[10] Salutation Consortium, "Salutation Architecture Specification," The Salutation Consortium Inc., available at ftp://ftp.salutation.org/salute/sa20e1a21.ps June 1, 1999.
[11] E. Guttman, C. Perkins, J. Veizades, and M. Day, "Service Location Protocol, Version 2," available at http://www.ietf.org/rfc/rfc2608.txt, June 1999.
[12] M. Nidd, "Service Discovery in DEAPspace," IEEE Personal Communications, pp. 39-45, 2001.
[13] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The design and implementation of an intentional naming system," presented at 17th ACM Symposium on Operating Systems Principles (SOSP '99), Kiawah Island, SC, 1999.
[14] M. Balazinska, H. Balakrishnan, and D. Karger, "INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery," presented at Pervasive 2002 - International Conference on Pervasive Computing, Zurich, Switzerland, 2002.
[15] R. Rivest and B. Lampson, "SDSI---A Simple Distributed Security Infrastructure," 1996, available at http://theory.lcs.mit.edu/~rivest/sdsi10.html.
[16] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, "SPKI Certificate Theory," 1999, available at http://www.ietf.org/rfc/rfc2693.txt.
[17] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," IEEE/ACM Transactions on Networking, vol. 8, pp. 281-293, 2000.
[18] F. Stajano and R. Anderson, "The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks," presented at 7th International Workshop on Security protocols, Cambridge, UK, 1999.
[19] F. Stajano and R. Anderson, "The Resurrecting Duckling -- what next?," presented at 8th International Workshop on Security protocols, Cambridge, UK, 2000.
[20] R. Want, T. Pering, G. Danneels, M. Kumar, M. Sundar, and J. Light, "The Personal Server - Changing the Way We Think about Ubiquitous Computing," presented at 4th International Conference on Ubiquitous Computing, Goteborg, Sweden, 2002.
[21] F. Stajano, Security for Ubiquitous Computing: John Wiley & Sons, LTD, 2002.
[22] A. Menezes, P. v. Oorschot, and S. A. Vanstone, Handbook of Applied Cryptography: CRC Press, 1996.
[23] F. Zhu, M. Mutka, and L. Ni, "Splendor: A Secure, Private, and Location-aware Service Discovery Protocol Supporting Mobile Services," presented at 1st IEEE Annual Conference on Pervasive Computing and Communications, Fort Worth, Texas, 2003.
[24] iButton home page, available at http://www.ibutton.com/.