© ACM, 1996. This is the authors' version of the work. It is posted here by permission of ACM for your personal use.
Not for redistribution. The definitive version is available at http://doi.acm.org/10.1145/238168.238184.

The Ω Key Management Service

Michael K. Reiter Matthew K. Franklin John B. Lacy Rebecca N. Wright

AT&T Bell Laboratories, Murray Hill, New Jersey, USA
{reiter,franklin,lacy,rwright}@research.att.com

Abstract

In this paper we introduce Ω, a distributed public key management service for open networks. Ω offers interfaces by which clients can register, retrieve, and revoke public keys, and escrow, use (to decrypt messages), and recover private keys, all of which can be subjected to access control policy. Ω is built using multiple servers in a way that ensures its correct operation despite the malicious corruption of fewer than one-third of its component servers. We describe the design of Ω, the protocols underlying its operation, performance in our present implementation, and an experimental application of the service.

1 Introduction

Key management remains the primary obstacle to the wide-scale use of cryptography. While numerous approaches to key management have been proposed for specific application domains, in our opinion few exhibit sufficient power and flexibility to support the full range of applications emerging today. Solutions relying on an off-line certification authority tend to support a limited set of functions and only very static (and thus potentially stale) key-to-principal bindings, due primarily to the unavailability of the certification authority. Solutions employing an on-line key management server, on the other hand, tend to suffer from a well-known tradeoff between security and availability, namely that replicating services for availability makes them more difficult to secure [13, 19, 12, 27].

In this paper we introduce Ω ("Omega"), a key management service for open networks whose goal is to provide flexible and powerful interfaces to meet the demands of an ever-widening range of applications. Ω provides the flexibility of an on-line server without incurring the fault-tolerance or security vulnerabilities usually associated with such servers. Ω supports interfaces by which a client can, if access control policy allows, (i) register a public key at the service, (ii) retrieve a public key that was registered at the service, (iii) revoke a registered public key from the service, (iv) escrow a private key at the service, (v) decrypt a message using a private key escrowed at the service, and (vi) fully recover a private key that was escrowed at the service. A goal of Ω is to provide a set of policy-independent functions that can be tailored to fit a wide range of key-management policies. So, for example, the escrow function provided by the service can be tailored for policies that *require* key escrow (e.g., in support of law enforcement) or that simply suggest it to enable a client to recover its private key in the event of its loss. Ω supports the described interfaces for both RSA [30] and ElGamal [7] keys.

Ω is a distributed service, built using multiple servers in a way that ensures its continued and correct operation despite the benign failure or even malicious penetration of up to a threshold number t of its servers, provided that the total number n of servers satisfies $n \geq 3t + 1$. This guarantee applies to all functions supported by the service. So, the penetration of up to t servers will not enable the attacker to, for example, alter the public keys distributed by the service, recover private keys that are escrowed at the service, or prevent the recovery of escrowed private keys by the proper authorities. Ω thus compensates for the increased difficulty of securing replicated servers by tolerating a failure to adequately protect some of them.

Ω is novel also in its key escrow functions, which are not typical of key management services. The motivation to support key escrow in Ω is twofold. First, escrow as a form of "key backup", so that a key can be recovered if it is lost, is a prerequisite for the use of strong encryption in some settings. Second, some applications require key escrow to enable protected and auditable use of a key in emergency situations (e.g., in a business or law-enforcement emergency) when the key would otherwise be unavailable. The escrow functions of Ω ensure that the private key corresponding to a public key being distributed by the service is escrowed at the service, and thus that (i) the private key can be recovered if, e.g., it is lost by its owner and (ii) messages encrypted under the public key can be decrypted by the service at the request of proper authorities, but without revealing the private key. Property (ii) should suffice, e.g., to enable authorities to recover a shared session key established between two clients with their public keys, if the clients conform to a known protocol by which that session key is established. However, if two clients deviate from the protocol for using public keys in certain agreed-upon ways (e.g., see [17]), or if they communicate without the aid of these keys, then these escrowed keys may be useless for monitoring client communication.

As described previously, a focus of our effort is to ensure that Ω can support a wide range of applications. To this end, we have engineered Ω to adapt easily to accommodate multiple certificate formats. In particular, Ω currently can produce X.509-compliant RSA certificates, and thus can interoperate with many existing and emerging applications. An Ω public key has been integrated into the Netscape™ World Wide Web browser beginning with version 1.1, to enable us to experiment with Internet applications. We have

.Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.
CCS '96, New Delhi, India
© 1996 ACM 0-89791-829-0/96/03..$3.50

also established Ω as a certification authority for World Wide Web servers within AT&T.

This paper provides an overview of Ω with a focus on its design and the protocols that underlie its operation. We begin by placing Ω in the context of prior work on key management systems in Section 2. Section 3 describes principles and protocols underlying the service. Section 4 presents the protocols that support public key registration, lookup, and revocation, and Section 5 describes the protocols for private key escrow, use, and recovery. Access controls to govern the use of these functions are discussed in Section 6. Two implementation issues, namely server recovery and performance, are discussed in Section 7. Section 8 describes ongoing efforts to apply Ω in the context of the Internet.

2 Related work

Ω has been most directly influenced by the first author's prior work on a fault-tolerant authentication substrate for the Horus system [28]. That work included the implementation of a prototype key distribution service with a similar architecture to that of Ω. However, that service did not support key escrow, supported only the distribution of RSA keys, was suited for use primarily in the Horus environment (e.g., it did not produce certificates that complied with emerging standards), and reached a level of maturity and performance far short of our present goal.

Ω's architecture differs from those of the key distribution services in the Digital Distributed System Security Architecture and its derivatives [11, 32, 19], the CCITT X.509 recommendation [15], Privacy Enhanced Mail [16], and the Yaksha public key extension to Kerberos [10]. In their simplest form, these services consist of an off-line *certification authority* that creates public key certificates, and an on-line directory that distributes these certificates to clients. In contrast, Ω is on-line and thus can provide more timely service to clients (e.g., can create certificates with shorter lifetimes, which simplifies timely revocation [28]). In addition, whereas the certificate-signing private key is protected by keeping it off-line in these prior approaches, Ω protects the private key by dividing it among multiple servers using cryptographic techniques. There are other key management services that adopt an on-line approach, such as Kerberos [23] and the Sesame public key extension to Kerberos [20]. These services, however, do not provide the fault and penetration tolerance of Ω. Methods to increase the fault and penetration tolerance of shared-key distribution services such as Kerberos have been proposed (e.g., [6, 12, 3]) but do not immediately extend to support the functions of Ω.

Ω's escrow techniques were most directly influenced by work on threshold signatures [5] and verifiable secret sharing [24]. Our escrow protocols employ these techniques to divide a private key among the servers, so that $t+1$ servers can collectively decrypt messages encrypted under the corresponding public key or fully reconstruct the private key, but t or fewer cannot. Mathematically these protocols resemble several other escrow methods proposed in the scientific literature (e.g., [21, 8, 17]), though minor differences arise due to differences in goals and system constraints. There is also a body of more distantly related proposals for key escrow, including the EES program [22], which provides symmetric key escrow using very different mechanisms. We know of no efforts besides Ω, however, that demonstrate penetration-tolerant key escrow, use, and recovery in practice.

3 State machine replication & Rampart

As described in Section 1, Ω guarantees its correctness despite the failure or corruption of up to a threshold t out of n servers provided that $n \geq 3t+1$. Basic to this guarantee is Ω's use of *state machine replication* [31] to mask the behavior of corrupt servers. State machine replication is a general technique for implementing fault-tolerant services using multiple identical, deterministic servers, each initialized to the same state. Client requests are issued to the service using an *atomic multicast* protocol, which ensures that each correct server receives the same sequence of requests. So, by correct servers processing requests in the order of receipt, they will all respond with the same output for each request. Provided that at most t servers are corrupt, the responses of corrupt servers can be masked by *output voting*, i.e., accepting only responses output by at least $t+1$ servers.

Ω makes use of state machine replication and, in our present implementation, of the Rampart toolkit [26] for this purpose. Rampart provides client-resident and server-resident modules to which application client and server programs interface. These modules combine to communicate client requests to a service via atomic multicast, and service responses to clients via output voting. Rampart's atomic multicast protocol [25, 26] tolerates the benign or malicious failure of t out of $n \geq 3t+1$ servers[1] and any number of clients. Rampart also enables servers to send authenticated atomic multicasts to the group of servers, provides a mechanism to detect a faulty server that does not multicast a message for which others are waiting, and ensures that correct servers concur on the set of messages multicast by such a server prior to its failure. Because our protocols require that Ω servers sometimes block awaiting atomic multicasts from other servers, these features are useful to ensure that our protocols will make progress.

The output voting protocol of Rampart is also well suited to support Ω; more accurately, it was redesigned to support Ω. This protocol is based upon a technique described in [27] for performing output voting using a *threshold signature scheme* [5]. Briefly, a $(t+1, n)$-threshold signature scheme is a technique for creating a public key and n *shares* of the corresponding private key in such a way that given a message, each share can be used to produce a *partial result* for that message, and any $t+1$ partial results for that message can be combined into the digital signature for that message. Moreover, $t+1$ shares are necessary to create the signature for the message, in the sense that without the private key, it is infeasible to produce (i) the signature from t or fewer partial results for the message, (ii) a partial result without the corresponding share, or (iii) another share from t or fewer shares. So, if one share is given to each server, at most t servers are corrupted, and each correct server generates partial results only for responses that it computes, then only those responses will ever be properly signed. The Rampart protocol currently employs an RSA-based threshold signature scheme due to Desmedt and Frankel [5].

The novelties of the Rampart output voting protocol that were driven by Ω derived from a combination of the needs for Ω to produce responses efficiently, to generate responses that conform with standards, and to enable clients to detect the misdirection or replay of responses from the service. In

[1] More precisely, our multicast protocols, which employ timeouts in their methods for failure detection, satisfy the described properties provided that messages from correct servers induce timeouts in other correct servers sufficiently infrequently. See [25] for details.

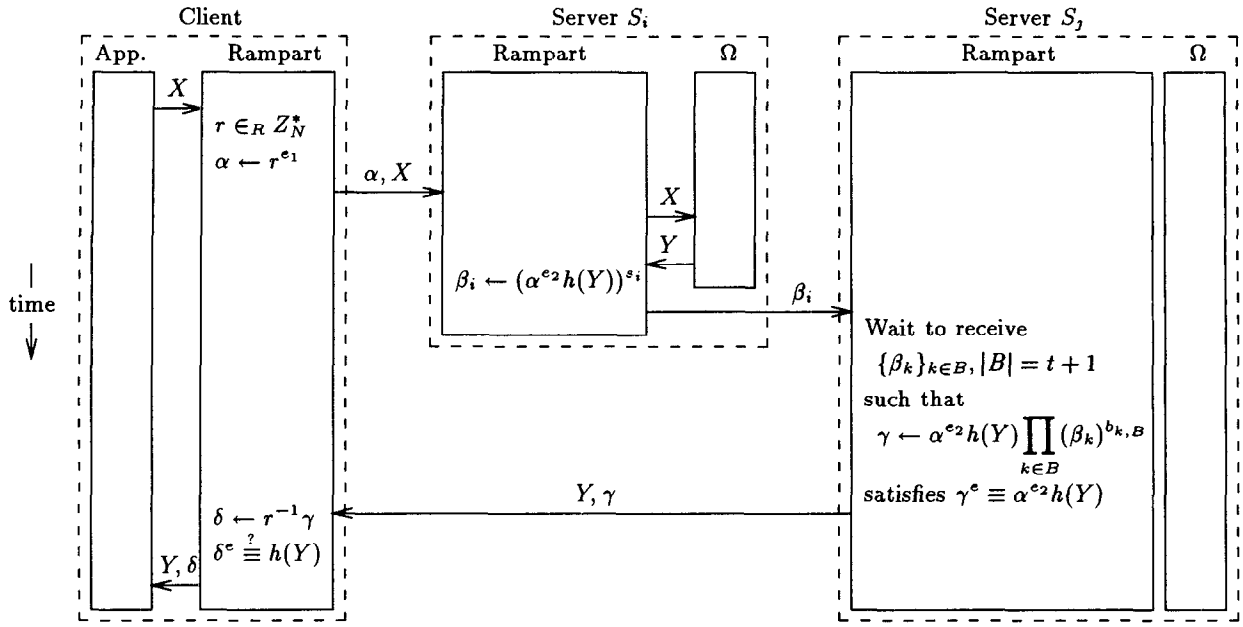

Figure 1: Rampart output voting protocol (operations performed mod N)

particular, among the interfaces offered by Ω is key lookup, to which Ω should return, e.g., an X.509 certificate. While this certificate, including its signature, could be generated by Ω over Rampart, this would incur the overhead of two signatures on the critical path of the reply: Ω signing the certificate and Rampart signing the response that contains the certificate. Instead, having the Rampart signature *be* the signature for the Ω certificate would be much more efficient. However, this precludes Rampart placing any material in the signed portion of the reply (e.g., the intended destination of the reply or a nonce identifier), because any such material would destroy the X.509 conformance of the reply.

For this reason, we redesigned the Rampart output voting protocol to prevent it from signing anything but the information provided by Ω, but so that the Rampart code on the client side could still detect a replayed or misdirected reply. To a first approximation, the Rampart client module accompanies each client request with a value ρ equal to a fresh random number $r \in Z_N^*$ encrypted under the RSA public key (e, N) of the service, i.e., $\rho = r^e \bmod N$. Once Ω has produced its reply Y, the Rampart server modules collectively generate a "blinded" [2] signature $(\rho \cdot h(Y))^d \equiv r \cdot h(Y)^d \bmod N$ for Y using the threshold signature scheme, where d is the private key of the service (shared among the servers) and h is a message digest function (e.g., MD5 [29]). On the client side, the Rampart module multiplies this value by $r^{-1} \bmod N$ to obtain $h(Y)^d \bmod N$, the signature for Y. Moreover, since no server or attacker learns r or $h(Y)^d \bmod N$, it is not feasible to undetectably substitute a replay or misdirect another reply Y' to the client.

A weakness exists in this protocol as described, because it provides a corrupt client the opportunity, via a chosen message attack, to obtain the service's signature on a message that the service did not intend to sign. That is, if the client sets $\rho = h(Y)^{-1} h(Y') \bmod N$, where Y is the expected response from the service and Y' is a response on which the client would prefer the service's signature, then the service will unexpectedly create $h(Y')^d \bmod N$. One way to remedy this would be to require the client to encrypt r using a chosen-ciphertext-secure version of RSA such as [9]. A slightly more efficient remedy, which we pursue here, involves presenting the service's public exponent e as a product $e = e_1 e_2$; we typically use $e_1 = 3$ and $e_2 = 5$. The computation of ρ is then done in two stages, one at the client and one at the service: the client chooses $r \in Z_N^*$ at random and sends $\alpha = r^{e_1} \bmod N$, and the servers generate $(\alpha^{e_2} h(Y))^d \bmod N$. This seems to force a client to take e_2-th roots mod N to mount chosen message attacks, which is equivalent to breaking RSA.

The full protocol is shown in Figure 1. This figure shows the operations performed at the client and each server, beginning with the client application generating its request X and ending with the client application receiving the reply Y from the service and the service's signature $\delta = h(Y)^d \bmod N$ for that reply. In particular, each operation contained in the box labeled "Server S_i" is performed by *each* server S_i and similarly for operations contained in the box labeled "Server S_j". The notation "$r \in_R Z_N^*$" denotes the choice of a random value $r \in Z_N^*$. The notation in this figure also reflects the threshold signature scheme currently in use by Rampart [5]. In this scheme, server S_i computes its partial result as $\beta_i = (\alpha^{e_2} h(Y))^{s_i} \bmod N$, where s_i is S_i's share of the private key d (or more accurately, of $d-1$) of the service. As partial results are received from other servers, each server waits until there is a set $B \subseteq \{1 \ldots n\}$, $|B| = t+1$, such that $\alpha^{e_2} h(Y) \prod_{k \in B} (\beta_k)^{b_{k,B}} \equiv (\alpha^{e_2} h(Y))^d \bmod N$, where each $b_{k,B}$ is a Lagrange coefficient that can be computed efficiently and in advance.

A disadvantage of this protocol is that it can be susceptible to replay attacks if the service ever gives the same

response twice. For this reason, each response from Ω is constructed to be unique, typically by including a unique value that can be ignored on the client side. A second disadvantage of this protocol is that while it enables the client to associate the response Y with the request received at the servers with α, the client has no assurance that its request X was not altered in transit. So, if necessary, Y must be of a form that allows the client to verify that it is an appropriate response to X. For example, if X is a request for a principal's public key, then Y should include the principal's name so the client can verify that it has obtained the public key for the intended principal.

4 Registration, lookup & revocation

In this section we describe the Ω protocols for handling public keys. These protocols enable clients to register, lookup, and revoke RSA or ElGamal public keys at the service. The Ω client-side module also provides functions to generate RSA or ElGamal key pairs, although these operations are local to the client and do not involve servers. Here we assume that the client issuing each request is authorized to perform the requested operation; access control is discussed in Section 6. Messages to servers are communicated by atomic multicast, and replies to clients are voted on using the protocol of Section 3.

Registration The Ω interface for registering public keys enables a client to submit a name and a public key to be stored together at the service. Before accepting such a pair, each server performs certain checks on the public key. If it is an RSA public key (e, N), where supposedly N is the product of two distinct primes, then each server verifies before accepting the registration that N is not a prime power (i.e., $N \neq p^k$ for any prime p and any $k > 0$). If it is an ElGamal public key (g, p, q, y), where supposedly p is prime, q is a (large) prime factor of $p-1$, g has order q in Z_p^*, and $y = g^z \bmod p$ for some z, then each server verifies these suppositions by checking that p and q are prime, that q divides $p-1$, and that $g^q \equiv y^q \equiv 1 \bmod p$. These verifications are performed primarily to simplify any subsequent escrow of the corresponding private key, as discussed in Section 5. If the key passes these verifications, then it is accepted and stored associated with the name.

Lookup To request a public key from Ω, the client specifies the name of the principal for which it is requesting a public key and the type of public key that it is requesting. Each server's response includes the most current public key of that type for that principal in its possession. In our present implementation, each server outputs the public key for the principal in an unsigned X.509 certificate, the signature for which is the signature that Rampart creates before sending the reply to the client (see Section 3).

Accommodating X.509 certificates prompted changes to the client request, because an X.509 certificate includes timestamps marking the lifetime of the certificate, which must be agreed upon at all correct servers. To facilitate this agreement, the client includes in its request a timestamp equal to the client's local clock value at the time of issuing the request. When each server receives this request, it verifies that the timestamp is sufficiently close to its own clock value and, if so, uses this timestamp as the base time from which to compute the certificate lifetime; this technique is described in more detail in [28]. Since each server uses its local clock, servers may disagree on whether the client's timestamp is sufficiently close to their clocks. However, by each server updating its state identically regardless of whether it replies with a certificate or a rejection message (i.e., by incrementing its certificate serial number counter in either case), this disagreement will not lead to divergence in server states. And, if at most t out of the $n \geq 3t+1$ servers are faulty, then a certificate or rejection message (or both) will be signed by the service and sent to the client. (If both, the second to arrive will be ignored by the client.)

Because the service produces a valid X.509 certificate, this reply can be used in conjunction with any application that requires certificates of this form, and in particular can be "pushed" [19] to other applications as is customary in many authentication protocols. (In Section 8, we describe such a use in the context of the World Wide Web.) Moreover, Ω is not bound to return only X.509 certificates, but can be easily adapted to generate other kinds of certificates from the information it stores. As additional certificate formats become widely used, we anticipate expanding the Ω interfaces to allow requests for multiple types of certificates.

Revocation The revocation of a principal's public key takes place by a client submitting a request to the service containing the name of the principal and the public key to revoke. Correct servers then no longer distribute that public key for that principal. Ω currently provides no interface to retrieve "revocation lists", in contrast to many other systems (e.g., [11, 16]). We feel that the need for such lists is diminished by the highly available nature of Ω: rather than retrieving a revocation list to see if a principal's key has been revoked, the client could just as well retrieve a new certificate for that principal. However, we anticipate providing an interface for retrieving revocation lists to be compatible with applications that require them.

The speed with which a revocation will propagate from Ω to the larger system it serves depends on how Ω is used. Ω can afford to create certificates with substantially shorter lifetimes (e.g., days, hours, or even minutes) than those usually associated with certificates created by an off-line certification authority, since Ω is available to provide fresh certificates on demand. As a result, an Ω certificate can be created with a prudent lifetime to ensure that it will expire "sufficiently shortly" after the revocation of the public key that it contains. We are currently examining other mechanisms to propagate revocations more quickly. One is a *callback* mechanism by which clients register interest in keys at Ω, and are informed by Ω when one of these keys is revoked.

5 Escrow, decryption & recovery

Ω supports the escrow of RSA and ElGamal private keys. Private key escrow ensures that messages encrypted under the corresponding public key can be decrypted by the proper authorities and that the private key can be recovered if, for example, it is lost by its owner. (Ω could be extended trivially to also support signing with escrowed private keys, though we haven't done so in our present implementation.) The correctness of these operations and the secrecy of the escrowed key are ensured despite the collusion of up to the threshold t of corrupt servers. Key escrow can be tied to key registration in Ω to provide leverage in enforcing escrow, by having the service refuse to distribute the public key until the corresponding private key has been escrowed.

When describing the following protocols, we again assume that the client issuing each request is authorized to perform the requested operation; access control is discussed in Section 6. The escrow protocols as presented here assume that a public key corresponding to the private key being escrowed has already been registered at the service. Moreover, our escrow protocols require the ability for the client to send private information to each server individually. To support this, Ω offers an interface by which a client can request a set of public keys, one for each server, to which each server replies with a set of public keys distributed among the servers at startup. Like other replies, this is voted upon using the output voting protocol of Section 3, and so the public keys obtained can be trusted. Below we assume that the client already possesses a public key K_i for each server S_i. We denote the encryption of v under public key K_i by $\langle v \rangle_{K_i}$, and the decryption of v with the private key K_i^{-1} by $\langle v \rangle^{K_i^{-1}}$. Again, all messages to servers (from clients or servers) are communicated by atomic multicast, and replies to clients are voted upon using the protocol of Section 3.

5.1 RSA

Escrow protocol Our RSA escrow protocol employs a threshold decryption scheme [5] in much the same way as the output voting protocol of Section 3 makes use of the same scheme for producing signatures. More precisely, a client escrows a private key of a principal at the service by breaking it into shares and distributing one share to each server. The servers verify that they collectively possess the private key by decrypting several "test messages" in the manner prescribed by the threshold decryption scheme. If the servers find that the partial results from $n-t$ servers consistently contribute to proper decryptions for all test messages, then the service accepts the escrow.

The protocol is shown more precisely in Figure 2. It is assumed in this figure that each correct server possesses a pair of values (e, N) that a client previously registered at the service as the principal's public RSA key. If that client was correct (and the public key was generated by the Ω client-side module), then $N = pq$, $p = 2p' + 1$, and $q = 2q' + 1$ for primes p, q, p' and q'. The client attempting an escrow should possess the private key d satisfying $ed \equiv 1 \bmod \lambda(N)$. Here, λ is the Carmichael function, i.e., $\lambda(N)$ is the least positive integer satisfying $m^{\lambda(N)} \equiv 1 \bmod N$ for all $m \in Z_N^*$. The escrow protocol ensures that the service can, if later presented with a message m by an authorized client, produce $m^d \bmod N$, or reveal d to an authorized client if necessary.

The protocol begins by the client choosing a random degree t polynomial $f(x) \in Z_{\lambda(N)}[x]$ satisfying $f(0) = d - 1$, and computing n shares $s_1, \ldots, s_n$ as $s_k = f(2k)/a_k \bmod p'q'$ where a_k is a value that is independent of e and N that can be computed efficiently and in advance. The client then sends $\{\langle s_k \rangle_{K_k}\}_{1 \le k \le n}$ to the servers by atomic multicast. Each server S_i does the following: it (i) decrypts $\langle s_i \rangle_{K_i}$ to obtain s_i, (ii) deterministically computes L "unpredictable" messages $\beta_1, \ldots, \beta_L \in Z_N^*$ by applying a message digest function h to L different values $V_1, \ldots, V_L$ known to the servers (each V_ℓ is described below), (iii) raises each β_ℓ to the es_i-th power mod N (i.e., computes its partial results for $\beta_1^e, \ldots, \beta_L^e$), and (iv) atomically multicasts these partial results to the other servers. Each server accepts the escrow attempt if there are $n-t$ servers whose partial results have the property that for all ℓ, $1 \le \ell \le L$, the partial results for β_ℓ^e from each subset of $t+1$ of them yield $(\beta_\ell^e)^d \equiv \beta_\ell \bmod N$ when properly combined. Given $n-t$ such partial results for β_ℓ^e, the verification that all $(t+1)$-subsets combine to form β_ℓ can be optimized by checking $n - 2t$ (carefully chosen) subsets of the $\binom{n-t}{t+1}$ subsets of size $t+1$.

The assurance of a proper escrow with this scheme derives from the following fact: since N is not a prime power (as verified in the registration protocol), a value $\delta \not\equiv 1 \bmod \lambda(N)$ satisfies $x^\delta \equiv x \bmod N$ for at most 1/4-th of the elements $x \in Z_N^*$. So, the probability that a subset of $t+1$ correct servers' partial results for some β_ℓ^e combine to form β_ℓ, even though their shares are invalid (i.e., their shares combine to form a value d' such that $ed' \not\equiv 1 \bmod \lambda(N)$), is at most 1/4. Thus, the probability that the partial results from a subset of $t+1$ correct servers that were given invalid shares satisfy this relationship for *all* ℓ, $1 \le \ell \le L$, is at most $1/4^L$. Since this holds for all $(t+1)$-subsets of a set of $n-t$ servers (t of which may be faulty), with high probability there are $n-2t$ correct servers that possess proper shares for a value d such that $ed \equiv 1 \bmod \lambda(N)$.

A limitation of this protocol is that it only works for RSA keys of the form $N = pq$ where $p = 2p'+1$, $q = 2q'+1$, and all of p, q, p', and q' are prime, because the threshold decryption scheme we use [5] works only for keys of this form. Finding such N in our implementation takes substantially more time than finding RSA moduli of a less restricted form, and many applications do not produce RSA keys of only this form. An alternative without this limitation is to share d to each $(t+1)$-subset of the servers separately with a $(t+1, t+1)$-threshold decryption scheme for general RSA keys (e.g., [1]), and to perform checks similar to those above for each $(t+1)$-subset. Though costly in general, this is only marginally more costly for small n and t (e.g., $t = 1$, $n = 4$). De Santis et al., have also proposed an escrow scheme for RSA without this limitation, and that is as secure as RSA [4].

Without special-purpose hardware at the servers for performing modular exponentiation, our protocol's performance is limited by L because, e.g., each server S_i must compute L partial results $(\beta_\ell)^{es_i} \bmod N$, $1 \le \ell \le L$. Each such exponentiation is costly, taking roughly 385 milliseconds on a 50 MHz SPARCstation 20 for a 768-bit N. The size of L that is needed to ensure a proper escrow is largely determined by the client's ability to predict, possibly with the help of corrupt servers, the test messages $\{\beta_\ell\}_{1 \le \ell \le L}$ that will be used to verify the escrow attempt. If these messages can be predicted far in advance, then L must be large enough to make a brute-force attack by the client infeasible (e.g., $L \ge 32$). However, this predictability can be limited in practice with simple tricks, such as computing each β_ℓ from information V_ℓ that includes, in addition to ℓ: (i) the encrypted shares that the client sends to the servers, (ii) the placement of the escrow request in the atomic multicast receipt sequence, (iii) the most recent prior atomic multicast from each server, (iv) the request immediately preceding the escrow request, etc. Assuming that these tricks render $\{\beta_\ell\}_{1 \le \ell \le L}$ unpredictable, we typically choose $L = 5$, leaving the client roughly a .001 probability of fooling the service. Further steps could be taken to ensure that $\{\beta_\ell\}_{1 \le \ell \le L}$ are unpredictable: e.g., each V_ℓ could include random numbers atomically multicast by servers, either periodically or per escrow request. We do not take such steps in our present implementation.

Decryption using escrowed keys Ω provides an interface by which a client can request that a message be decrypted with a private RSA key escrowed at the service. When presented with a message m to be decrypted with an escrowed private key, each server S_i looks up the corresponding public key (e, N) and its share s_i of the private key and

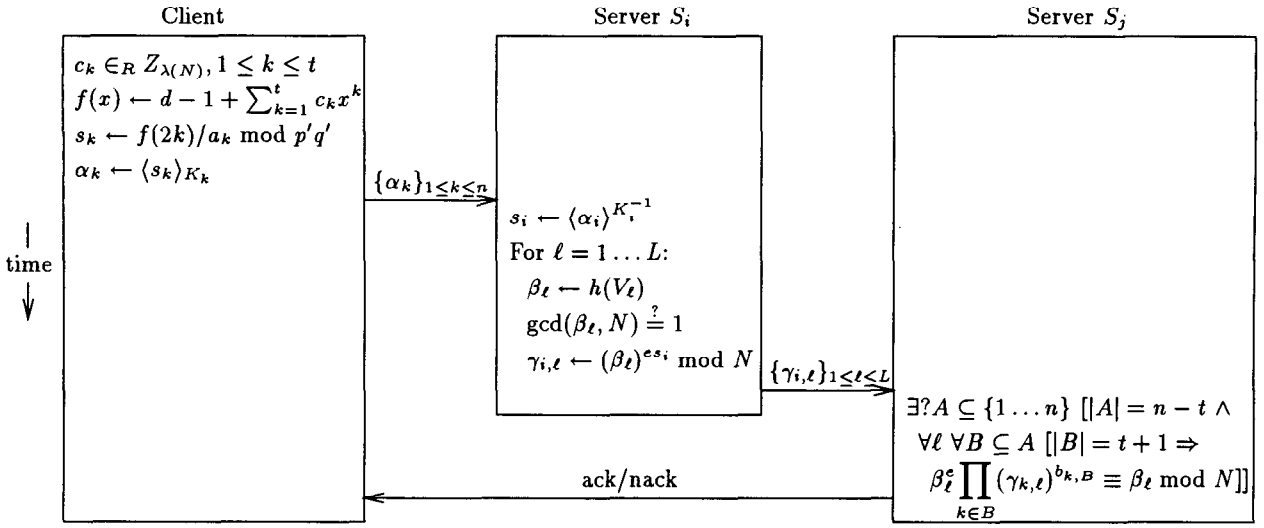

Figure 2: RSA key escrow

decrypts m as prescribed by the threshold decryption algorithm. That is, S_i computes its partial result $m^{s_i} \bmod N$, atomically multicasts this to the other servers, waits to receive partial results that allow it to reconstruct an m' such that $(m')^e \equiv m \bmod N$, and replies to the requesting client with m'. Since $n - 2t$ servers are guaranteed (with high probability) to possess correct shares of the private key at the end of the escrow protocol, such an m' will be found provided that $n \ge 3t + 1$.

This protocol as described would allow corrupt servers and network eavesdroppers to learn m'. To hide this plaintext, the client that issues the ciphertext to be decrypted first blinds the ciphertext before submitting it, by multiplying it by $r^e \bmod N$ for a random $r \in Z_N^*$. Upon receiving the decrypted reply, it multiplies the result by $r^{-1} \bmod N$ to obtain the target plaintext.

Recovery of escrowed keys Ω also provides an interface by which a client can request a private RSA key escrowed at the service. In response to such a request, each server S_i encrypts its share s_i of the private key d under a public key K provided in the client's request and atomically multicasts this value to the other servers. Each correct server then responds to the client with the collected set of encrypted shares. The client decrypts these shares with K^{-1} and determines d via Lagrange interpolation.

5.2 ElGamal

Escrow protocol Our ElGamal escrow protocol enables a client to escrow the private key z corresponding to a public key $(g, p, q, y = g^z \bmod p)$ that was previously registered at the service. The protocol assumes that p is prime, q is a prime factor of $p-1$, g has order q in Z_p^*, and y is generated by g. Recall that each of these assumptions was verified when (g, p, q, y) was registered at the service (see Section 4).

Like our RSA escrow protocol, our ElGamal escrow protocol adapts a prior protocol, initially designed for a somewhat different purpose, to achieve key escrow. In this case, the original protocol is for sharing the discrete logarithm of a public value and is due to Pedersen [24]. The escrow protocol is shown in Figure 3. The protocol begins by the client, which possesses the private key z, choosing a random degree t polynomial $f(x) \in Z_q^*[x]$ such that $f(0) = z$; denote f by $f(x) = c_t x^t + \ldots + c_1 x + z$. The client then creates and sends an escrow request consisting of $\{\langle s_k \rangle_{K_k}\}_{1 \le k \le n}$, where $s_k = f(k) \bmod q$, and $\{\gamma_k\}_{1 \le k \le t}$, where $\gamma_k = g^{c_k} \bmod p$. Each server S_i individually verifies that $g^{s_i} \equiv y \prod_{k=1}^{t} (\gamma_k)^{i^k} \bmod p$, and atomically multicasts "ok" to the other servers if this check succeeds (and sends "nok" otherwise). Finally, each server accepts the escrow attempt if "ok" was received from at least $n - t$ servers. If the correct servers accept this escrow attempt, then $n - 2t$ correct servers possess a correct share of z (see [24]).

Decryption using escrowed keys Ω provides an interface by which a client can request that a message be decrypted with a private ElGamal key escrowed at the service. This is implemented at the servers with a protocol for computing $\alpha^z \bmod p$ where α is a value provided by the client, z is the private key escrowed at the service, and (g, p, q, y) is the corresponding public key registered at the service. This suffices to support ElGamal decryption because the ElGamal encryption of a message m under a public key (g, p, q, y) is $(g^k \bmod p, m y^k \bmod p)$ where k is a random element of Z_q^*. So, if a client submits to Ω the first component α of a ciphertext (α, β) generated with key (g, p, q, y), and Ω returns $\alpha^z \bmod p$, then the client can find the plaintext $m = (\alpha^z)^{-1} \beta \bmod p$.

To compute $\alpha^z \bmod p$ for a client that provides α, each server S_i that sent "ok" when z was escrowed looks up its share s_i for z, and atomically multicasts its partial result $r_i = \alpha^{s_i} \bmod p$ to all servers. Each server then forms its reply to the client as follows. Let $\{r_k\}_{k \in A}$ for some $A \subseteq \{1, \ldots, n\}$ be the partial results received at all servers.

1. If there is some $B \subseteq A$, $|B| = 2t + 1$, such that for all $B' \subseteq B$ of size $t+1$, the computation $\prod_{k \in B'} r_k^{b_{k,B'}} \bmod$

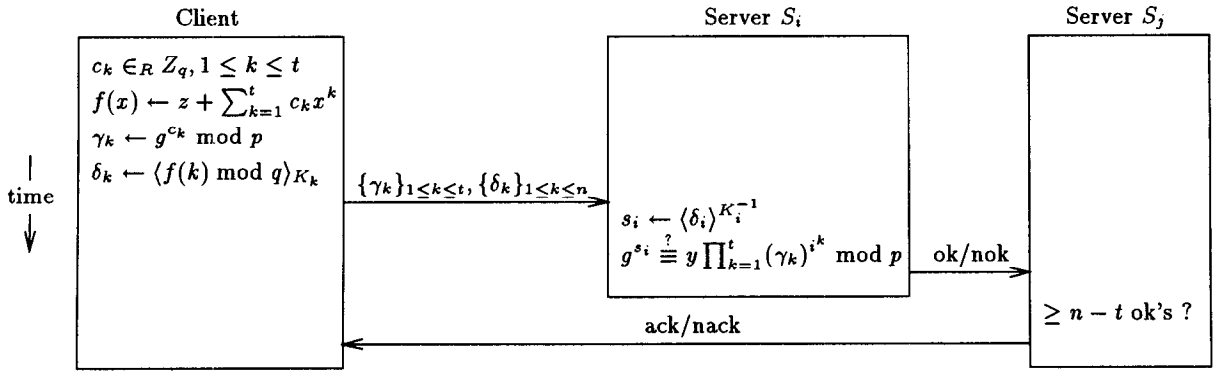

Figure 3: ElGamal key escrow

p, where $b_{k,B'}$ is the appropriate Lagrange coefficient, produces the same value, then the server replies to the client with this value. This value is $\alpha^z \bmod p$, because the partial results $\{r_k\}_{k \in B}$ contain partial results from $t+1$ correct servers that sent "ok" in the escrow protocol. However, since at the end of the escrow protocol only $n-2t$ correct servers are guaranteed to possess proper shares of z, there may be no such set B if $n < 4t+1$.[2]

2. If there is no such set B, then the server replies with

$$\left\{ \prod_{k \in B'} r_k^{b_{k,B'}} \bmod p \right\}_{B' \subseteq A, |B'| = t+1}.$$

(Note that this set will contain at most $\binom{3t}{t+1}$ elements.) Since $n \ge 3t+1$ and at least $n-2t \ge t+1$ correct servers completed the escrow protocol with valid shares of z, the value $\alpha^z \bmod p$ is contained in this set. However, it is left to the requesting client to determine which value is $\alpha^z \bmod p$ (e.g., by trying to complete the ElGamal decryption with each value, provided that the target plaintext is recognizable).

This protocol as described would allow corrupt servers and network eavesdroppers to learn the target value $\alpha^z \bmod p$ and thus the plaintext m corresponding to the target ciphertext $(\alpha, m\alpha^z \bmod p)$. To hide $\alpha^z \bmod p$, the client blinds α by submitting $\alpha^v \bmod p$ for some random $v \in Z_q^*$. Upon receiving $\alpha^{vz} \bmod p$ from the service, the client computes $\alpha^z \equiv (\alpha^{vz})^{v^{-1} \bmod q} \bmod p$.

Recovery of escrowed keys Ω also provides an interface by which a client can request a private ElGamal key escrowed at the service. In response to such a request, the service replies with a set of shares for that private key as in the recovery protocol for an RSA key, i.e., each share encrypted with a public key K included in the recovery request. The client decrypts these shares and determines the private key via Lagrange interpolation.

[2] By appending to the escrow protocol an additional interaction with the client, the existence of such a B can be guaranteed whenever $n \ge 3t+1$.

6 Access control

So far we have described what the service can do, but not for whom the service will do it. The latter is determined by the *access control policy* that describes what operations each client is authorized to perform. This policy is essential to the semantics of the keys managed by the service. If, for example, any client is allowed to register any public key for any principal, then public keys retrieved from the service are meaningless. In our present implementation, Ω enforces a few simple policies derived from the needs of applications with which we have experimented. Below we sketch a few of these policies, enforcement mechanisms and simple alternatives, for illustrative purposes only. Ω can be adapted with little effort to enforce more sophisticated policies.

Public key registration Since presumably clients will use a public key retrieved from Ω to authenticate the principal named with that key, it is important that Ω authenticate a client submitting a registration request as acting on behalf of the principal named in its request. Of course, it is not possible to require digital signatures to authenticate registration requests, as typically the registration of a public key for a principal precedes the service's possession of a public key for that principal. Rather, our present implementation presumes an out-of-band negotiation that results in a message digest of a principal's public key being stored at each server as a prerequisite to a client registering a public key on behalf of that principal. This supports a registration scenario in which the principal generates its (potentially large) public key and private key in isolation, computes a short message digest of the public key (e.g., 16 or 32 hexadecimal digits), and communicates this digest to one or more administrative authorities that authenticate the principal and install the principal's name and digest at each server. Each server accepts a registration request only if the digest of the public key in the request matches the stored digest for the principal named in the request. An example of such a registration scenario is described in Section 8.

Public key revocation To prevent public keys from being revoked capriciously, Ω restricts which clients can revoke each public key. Our present implementation requires a client to possess the corresponding private key. That is,

each revocation request is signed with the private key corresponding to the public key being revoked, in order to prove the client's authority to revoke this public key. In this way, only the owner of the key (or one who has compromised the private key) can revoke it.

Private key escrow Ω enforces no policy regarding which clients can escrow which private keys. Rather, any escrow that succeeds is assumed to imply knowledge of the private key by the client that issued the escrow request. Greater assurance of this could be obtained by requiring the client to sign its escrow request with the private key being escrowed.

Private key recovery Since private key recovery is offered primarily for those cases in which the private key is unavailable to the client, determining a client's authority to recover a private key should not depend on the client's ability to sign its request with that private key. Rather, this authorization can be determined with the aid of an "out-of-band" mechanism similar to that described above for key registration, i.e., that results in a message digest of the public key K in the recovery request being stored at the service. This supports a recovery protocol in which the principal generates the key K for the purpose of recovering its original key and communicates K's digest to administrators out-of-band. Alternatively, Ω could provide an interface for a client to specify, prior to the loss of a principal's private key, values of K for which the service should participate in the recovery protocol for that principal's key. This interface could authenticate the client by requiring a signature with that principal's key. Suitable values of K might be, e.g., public keys of other principals that the principal trusts, thus enabling the (auditable) recovery of its key by these principals.

7 Implementation issues

At the time of this writing, an initial research prototype of Ω is nearing completion. This implementation employs the Cryptolib toolkit [18] for its basic cryptographic operations and, as described in Section 3, the Rampart toolkit for atomic multicast and output voting in support of state machine replication. In this section, we briefly discuss two issues surrounding this implementation.

7.1 Logging and server recovery

Each Ω server maintains a log recording the sequence of operations that it performs, except for public key lookups. Each record contains the essential portions of the client request that invoked the operation, any follow-up messages from other servers (in the case of private key operations), and status information. The logs contain only public data; private data is stored in a separate data structure. Since messages to servers are communicated by atomic multicast and the servers are deterministic (see Section 3), the logs at all correct servers are identical.

The primary purpose of this log is to assist in the recovery of a server that failed. When a server recovers, each correct server communicates to the recovering server the portions of its log that will enable the recovering server to operate as if it had never failed. These portions include, among other things, records of public key registrations and revocations, and of private key escrow operations. The integration of a new server into operation is similar, but is complicated by the fact that an escrow operation prior to the new server's installation will include no share of the escrowed private key for the new server. Thus, subsequent decryptions using the escrowed key will not involve the new server.

Log information is communicated to the recovering server via the *state transfer* mechanism of Rampart. Rampart informs servers of a new or recovering server by inserting a special event in their atomic multicast delivery sequence. When the application servers receive this event, each can provide information for updating the new server to Rampart. Rampart delivers to the new server information provided by at least $t+1$ servers (to ensure that the new server is updated by only information from correct servers) and buffers requests to the new server until the server has had the opportunity to update its state.

Recovering or adding servers raises questions regarding the fault-tolerance of our service, which so far we have described statically as tolerating t failures out of n servers. This is the true tolerance of the service to *malicious* faults that expose secret values held by the faulty servers: the penetration of any $t+1$ servers by an attacker would, for example, enable the attacker to sign responses from the service. However, the actual tolerance of our service to *benign* (e.g., crash) failures is a more dynamic quantity that can exceed t out of n over the long term. Specifically, $\lfloor (n-1)/3 \rfloor$ is the maximum number of *concurrent* benign server failures that the service can tolerate and still make progress, if for no other reason than this is true of the Rampart protocols [25]. However, more than $\lfloor (n-1)/3 \rfloor$ benign failures can be tolerated serially, and in general all servers can fail benignly at some point, provided that some have recovered before others fail. A caveat to this statement is that benign server failures can prevent the decryption of a message with a key escrowed at the service, or the recovery of that key, until enough servers possessing proper shares for that key recover (recall that there are at least $n-2t$ of them, where n is the number of servers at the time of escrow). This circumstance will force the requesting client to reissue its request later, but will not deadlock the service. A direction of ongoing work is to adapt Ω to tolerate greater numbers of malicious failures that expose secret values held by the faulty servers, perhaps serially as for benign failures.

7.2 Performance

We anticipate that the performance of Ω will not be a limiting factor for most applications, for two reasons. First, in the applications that we envision, a typical client would employ all but perhaps the public key lookup operation infrequently. Second, lookups can be performed off the critical path of many protocols when performance is of concern (see, e.g., [28]). Nevertheless, understanding the factors that limit the performance of Ω is essential to determining its ability to scale to large numbers of clients and its suitability for use with certain protocols. In this section, we discuss its performance based upon experiments with our research prototype.

Preliminary performance numbers for the operations described in Sections 4 and 5, in the absence of faulty clients or servers, are shown in Table 1. These numbers are mean round-trip latencies in milliseconds (ms), as timed by the Ω client from initiating the operation to receiving the service's reply. The preparation of requests and the verification of the signature on the reply are included in these latencies, but the access controls described in Section 6 are not. In these tests, the client and server processes were each running on a separate SPARCstation 20 workstation. There were four

servers in these tests. All keys (notably the service's public key) contained 768-bit moduli. Keys used to encrypt shares in the private key escrow and recovery protocols were RSA keys, regardless of the client key type.

Operation	Client key type	
	RSA	ElGamal
public key registration	1884	756
public key lookup	563	575
public key revocation	588	595
private key escrow	3648	1049
private key decryption	1365	1330
private key recovery	1357	1165

Table 1: Mean latency (ms); 768-bit moduli, SPARC 20s

With the described key sizes, the mean round-trip latency of a null operation at the service was roughly 550 ms, over three-fourths of which was due to the modular exponentiation operations of the threshold signature scheme used to sign responses (see Section 3). The remainder of this time resulted primarily from costs associated with communication, particularly the atomic multicast protocol of Rampart. However, since the latency of this protocol is also partly due to modular exponentiations (see [26]), modular exponentiation is responsible for a large majority of the latency of a null request to the service. Even in most operations for which the 550 ms latency of the basic round-trip protocol was not the dominant cost, modular exponentiation continued to dominate the total latency. Specifically, our escrow, decryption, and ElGamal registration protocols, as well as the atomic multicast protocols that underlie them, employ modular exponentiation heavily at the servers.

The only operation whose cost was *not* dominated by modular exponentiation is the RSA registration protocol. As described in Section 4, one step of this protocol is for each server to verify that N, the public key modulus provided by the client, is not a prime power. This is done by verifying that N is not a proper power of *any* integer, which is performed by taking k-th roots of N for each prime k, $1 < k \leq |N|$, where $|N|$ is the bit length of N. Finding k-th roots of N, which is performed with Newton's method, dominated the latency of the registration protocol. This verification could be skipped in the registration protocol, although doing so would require either doubling the value L in the RSA escrow protocol to achieve the same level of security or, in the event that an attempted decryption with the escrowed key failed, factoring the modulus of the escrowed key (which, with high probability, is a prime power).

The conclusion that we draw from our preliminary performance experiments is that Ω is a compute-intensive service. Most computation takes place at the servers (not the clients) and takes the form of modular exponentiations. Equipping the servers with special-purpose hardware for performing modular exponentiation would dramatically improve the performance of the service and its ability to scale to large numbers of clients. Similarly, the performance of the service should improve substantially by employing modern server machines with more powerful processors. Finally, because most computation is performed at the servers, we expect that the service will be usable by a wide range of client devices. Further experiments, however, are needed in this area.

8 Applications: an example

In a first step towards experimenting with applications, we arranged for an Ω public key to be included in the Netscape World Wide Web (WWW) browser, beginning with version 1.1. This enables these browsers to accept certificates issued by Ω when interacting with WWW servers that support the SSL protocol [14]. More precisely, when a browser communicates with a WWW server, the server can authenticate itself to the browser by sending an X.509 certificate binding the server's name to a public key. If the browser possesses a public key with which it can verify the signature on that certificate, then it subsequently authenticates the server with the public key in the certificate, or more accurately, with a shared encryption key established using this public key. Thus, by having an Ω public key included in the browser, it is possible for browsers to authenticate and communicate privately to servers certified by Ω.

The bulk of the effort required to integrate Ω to work with Netscape browsers has been to establish Ω as a source from which a WWW server administrator can obtain a certificate for its server. Typically, a server administrator obtains a certificate for its server by generating a public/private key pair, and sending the public key and naming information via electronic mail to a certificate issuer. After receiving this request, the issuer takes measures (e.g., via a phone call) to authenticate the requesting administrator and then returns a properly signed certificate via electronic mail.

To simplify this certification process, we developed a WWW interface that enables WWW server administrators within AT&T to communicate their certificate requests to an Ω administrator (i.e., one of us). When a request is received, the Ω administrator verifies the employment status and intent of the requesting administrator and installs the name of the WWW server and a message digest of the public key at each Ω server. Another WWW interface can then be used to register the public key at Ω and retrieve an X.509 certificate for the WWW server. We plan to automate more of this procedure over time, for example using on-line databases of AT&T employees to verify employment status.

Ω has been operating as a certification authority for WWW servers within AT&T for roughly three months at the time of this writing. More ambitious application of Ω, for example in the areas of electronic commerce, electronic mail, and secure networking, is a direction of ongoing work.

9 Conclusion

Though a number of approaches to key management have been proposed, we believe that few have been demonstrated that possess the flexibility and robustness required by emerging applications. The Ω service attempts to address this need. Ω provides a collection of key management functions—including public key registration, lookup, and revocation, and private key escrow, decryption, and recovery—that can be tailored to suit a wide range of key-management policies. Moreover, Ω is tolerant of even the malicious penetration of fewer than one-third of its servers. Our initial prototype implementation of the service indicates that Ω is a viable service for key management, particularly if the servers are equipped to perform modular exponentiation efficiently.

Our current focus is refining the implementation of the functions described in this paper. One direction of ongoing work is providing interfaces for managing the service, such

as interfaces for specifying access controls for each service operation. A second direction for future work is integrating Ω within a key management hierarchy and exploring other alternatives for scaling Ω to large numbers of clients.

Acknowledgements We thank Elizabeth Royer for implementing the WWW interface to Ω.

References

[1] C. Boyd. Digital multisignatures. In H. J. Beker and F. C. Piper, editors, *Cryptography and Coding*, pages 241–246. Clarendon Press, 1989.

[2] D. Chaum. Blind signatures for untraceable payments. In R. L. Rivest, A. Sherman, and D. Chaum, editors, *Proceedings of CRYPTO '82*, pages 199–203. Plenum Press, 1983.

[3] L. Chen, D. Gollman, and C. Mitchell. Key distribution without individual trusted authentication servers. In *Proceedings of the 8th IEEE Computer Security Foundations Workshop*, pages 30–36, June 1995.

[4] A. De Santis, Y. Desmedt, Y. Frankel, and M. Yung. How to share a function securely. In *Proceedings of the 26th ACM Symposium on Theory of Computing*, pages 522–533, May 1994.

[5] Y. Desmedt and Y. Frankel. Shared generation of authenticators and signatures. In J. Feigenbaum, editor, *Advances in Cryptology—CRYPTO '91 Proceedings* (Lecture Notes in Computer Science 576), pages 457–469. Springer-Verlag, 1992.

[6] Y. Deswarte, L. Blain, and J. Fabre. Intrusion tolerance in distributed computing systems. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 110–121, May 1991.

[7] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31(4):469–472, July 1985.

[8] Y. Frankel and M. Yung. Escrowed encryption systems visited: Threats, attacks, analysis and designs. Manuscript, Nov. 1994.

[9] M. K. Franklin and M. K. Reiter. Adaptive chosen ciphertext security for RSA from Guillou-Quisquater signatures. Manuscript, May 1995.

[10] R. Ganesan. Yaksha: Augmenting Kerberos with public key cryptography. In *Proceedings of the 1995 Internet Society Symposium on Network and Distributed System Security*, pages 132–143, Feb. 1995.

[11] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The Digital distributed system security architecture. In *Proceedings of the 12th NIST/NCSC National Computer Security Conference*, pages 305–319, Oct. 1989.

[12] L. Gong. Increasing availability and security of an authentication service. *IEEE Journal on Selected Areas in Communications*, 11(5):657–662, June 1993.

[13] M. P. Herlihy and J. D. Tygar. How to make replicated data secure. In C. Pomerance, editor, *Advances in Cryptology—CRYPTO '87 Proceedings* (Lecture Notes in Computer Science 293), pages 379–391. Springer-Verlag, 1988.

[14] K. E. B. Hickman and T. ElGamal. The SSL protocol. Internet draft, June 1995.

[15] International Telegraph and Telephone Consultative Committee (CCITT). *The Directory - Authentication Framework, Recommendation X.509*, 1988.

[16] S. T. Kent. Internet privacy enhanced mail. *Communications of the ACM*, 36(8):48–60, Aug. 1993.

[17] J. Kilian and T. Leighton. Fair cryptosystems, revisited. In D. Coppersmith, editor, *Advanced in Cryptology—CRYPTO '95* (Lecture Notes in Computer Science 963), pages 208–221. Springer-Verlag, 1995.

[18] J. B. Lacy, D. P. Mitchell, and W. M. Schell. CryptoLib: Cryptography in software. In *Proceedings of the 4th USENIX Security Workshop*, pages 1–17, Oct. 1993.

[19] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, Nov. 1992.

[20] P. V. McMahon. SESAME V2 public key and authorisation extensions to Kerberos. In *Proceedings of the 1995 Internet Society Symposium on Network and Distributed System Security*, pages 114–131, Feb. 1995.

[21] S. Micali. Fair public-key cryptosystems. In E. F. Brickell, editor, *Advances in Cryptology—Proceedings of CRYPTO '92* (Lecture Notes in Computer Science 740), pages 113–138. Springer-Verlag, 1992.

[22] National Institute of Standards and Technology. *Escrowed Encryption Standard*, Feb. 1994. Federal Information Processing Standards Publication 185, U.S. Department of Commerce.

[23] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9), Sept. 1994.

[24] T. P. Pedersen. Distributed provers with applications to undeniable signatures. In D. W. Davies, editor, *Advances in Cryptology—EUROCRYPT '91 Proceedings* (Lecture Notes in Computer Science 547), pages 221–242. Springer-Verlag, 1991.

[25] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, Nov. 1994.

[26] M. K. Reiter. The Rampart toolkit for building high-integrity services. In K. P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems* (Lecture Notes in Computer Science 938), pages 99–110. Springer-Verlag, 1995.

[27] M. K. Reiter and K. P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986–1009, May 1994.

[28] M. K. Reiter, K. P. Birman, and R. van Renesse. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 12(4):340–371, Nov. 1994.

[29] R. L. Rivest. *RFC 1321: The MD5 Message Digest Algorithm*. Internet Activities Board, Apr. 1992.

[30] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, Feb. 1978.

[31] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[32] J. J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 232–244, May 1991.