Vault: A Secure Binding Service

Guor-Huar Lu, Changho Choi, and Zhi-Li Zhang
University of Minnesota
luxx0137@umn.edu, {choi, zhzhang}@cs.umn.edu

***Abstract*— Binding services are crucial building blocks in networks and networked applications. A binding service (e.g., the Domain Name System (DNS)) maps certain information, namely, binding keys (e.g., host names), to other information, i.e., binding values (e.g., IP addresses), and answers queries for such *key-value* bindings. Clearly, building secure *binding* services that ensure the integrity and authenticity of bindings are vital to the correct operations of many networks and networked applications.**

In this paper we present a novel approach for building *generic* secure binding services that allow *arbitrary* key-value bindings as (trusted) *infrastructure services* to support a variety of networks and networked applications. We combine the *Identity-Based Encryption* (IBE) crypto-mechanisms with distributed hash table (DHT) techniques to develop an innovative architecture for building scalable, robust and secure binding services. Using this architecture, we implement a prototype system called *Vault* and evaluate its performance both in a local testbed and on the PlanetLab.

I. Introduction

Binding services are crucial building blocks in networks and networked applications. For example, the domain name system (DNS) binds host names to IP addresses, providing an indispensable infrastructure service to Internet applications. In (SIP-based) VoIP services, registrars/location servers are needed to map users' names (SIP URLs) to appropriate SIP proxy servers for signaling and other purposes. In general, a binding service translates some keys (*binding keys*) into corresponding values (*binding values*), i.e., each *binding* is a ⟨`key, value`⟩ pair associated with a user – the *owner* of the binding – that utilizes the service. In a sense, a binding system is a *specialized* look-up or directory service where the notion of *ownership* of binding is essential: each binding has an owner (the client who registers the binding), and only the owner can update or delete the binding. In this paper we are especially interested in building a *generic* binding service – where we place *no* restrictions on the syntax or semantics of binding keys – in particular, the identity of an owner of a binding may not be part of the binding key[1] – as a (trusted) *infrastructure* service that can be shared by a variety of applications.

[1]The conventional DNS hostname to IP address binding is an example where the binding key, e.g., `host.foo.com`, (typically) contains the owner's identity, here `foo.com`, due to the hierarchical namespace of DNS. On the other hand, reverse DNS look-up is an example where the owner identity is not part of the binding key, here an IP address. Other examples include SIP registration which binds a phone number or an IP address to a SIP proxy, or location service (e.g., in wireless networks) that binds a phone number or an IP address to a geographic location. There are many other situations in real life where we want to bind certain keys to other pieces of information where either the owner identity is immaterial or concealed, while at the same time the authenticity of the bindings must be preserved.

Clearly, security is a key concern in building a binding service over an (untrusted) wide-area network, as integrity of bindings is often vital to the correct operations of networks and applications that rely on it. In building a (generic) *secure* binding service, there are two *minimum* requirements: i) as stated earlier, the service must ensure that only the owner of an existing binding can update or delete the binding; and ii) users of the service must have the guarantee that the bindings returned to their queries must indeed be the correct bindings deposited into the service by the owners. To meet these two requirements, a secure binding service must be robust against "man-in-the-middle" binding poisoning (e.g., DNS poisoning) attacks, where "imposters" interposed between users and the service cannot alter binding insertion requests from owners to the service or binding query replies from the service to the queriers. In addition, a (generic) secure binding service must also be scalable and robust. The combination of security, scalability and robustness makes building a secure binding service a challenging task.

In this paper we develop a novel approach for building *generic* secure binding services as (trusted) *infrastructure services* to support a variety of networks and networked applications that require secure binding as a core function. The generality of our secure binding services comes from the adoption of a *semantic-free, flat* "identifier" (*id* in short) space [1] as the intermediary for bindings. This allows us to separate the more complex, *policy* issues such as *namespace management* issues (e.g., name structure, ownership, etc.) from the basic *mechanisms* for building secure binding services. It also allows us to employ distributed hash table (DHT) techniques (e.g., [2], [3]) for building highly robust and scalable distributed systems. The key *novelty* of our approach lies in combining the *Identity-Based Encryption* (IBE) [4] with DHT techniques to develop a secure binding (infrastructure) service – referred to as *Vault*. Vault is robust against "man-in-the-middle" attacks and the security of Vault does not rely on *exogenous* trusted third-party such as PKI or requiring users to have their own public key pairs.

As an infrastructure service, Vault is built using a two-level architecture: at the core of the system are a number of special nodes (computer systems) called *pillars*, which are assumed to be highly fault-tolerant, and play a critical role in ensuring the security of the service, which includes *private* (decryption) key generation [4]; they do not directly interact with users nor store their bindings. These functions are performed by the rest of the nodes called *columns*, which form a ring structure using Chord [2] and may be added to or removed from the system dynamically. A user who wants to insert a binding or query for a binding encrypts its request using the *binding key id* (a hash of the key), and sends the encrypted requests to Vault; only corresponding node (column) in Vault that owns the key id can decrypt the request, perform the requested function, and reply to the user in a secure manner.

There are many challenging design and implementation issues in combining DHT and IBE to develop a scalable, robust and secure binding service. First and foremost, to provide the security and trustworthiness of Vault as an infrastructure service, we must ensure only authorized nodes (columns) can join Vault, and only active columns (those are part of Vault at a given time) can decrypt user requests, store bindings and answer queries. To address these issues, we develop an innovative mechanism called *secure bond* that allows a column to prove to pillars that they indeed own a requested key id so as to retrieve the corresponding decryption key. Furthermore, because IBE operations are computationally expensive, efficiency is also a major concern. In addition, although this paper focuses on the basic mechanisms for building secure binding services by separating the policy issues as "add-ons", we do take the importance of policy issues into account by providing the necessary interfaces and "plug-ins". In the remainder of this paper we present the design and implementation of Vault, and discuss how these issues are addressed. In the next section we present what is IBE and illustrate why IBE is uniquely suited for building secure binding services using DHTs. In Section III we explain the Vault architecture and operations as well as some further enhancements. In Section IV we evaluate the performance of a Vault prototype we have deployed both in a local area testbed and on the PlanetLab. The paper is concluded in Section V.

II. Background and Motivation

In this section we briefly explain what is identity-based encryption (IBE), and motivate why IBE is particularly suited for building secure binding services using DHT techniques with a flat identifier space.

A. Identity-Based Encryption

Originally proposed as a means to simplify certificate management in email systems [4], identity-based encryption (IBE) allows any arbitrary string (e.g., email address or other user identifiers) to be used as the public key. The corresponding private key is generated by a central authority (called the *private key generator*, or PKG in short). For example, if Bob wants to send a secret email to Alice, he encrypts the email using Alice's email address (i.e., her identity) as the public key. For decryption, Alice first retrieves her private key from the PKG and subsequently uses it to decrypt the email. Hence in IBE, as long as Bob knows the identity (here email address) of Alice, he can send an encrypted email to Alice. Whereas, in the conventional public-key cryptosystems such as RSA, Bob first needs to obtain the public key of Alice, and has a way to ensure that the public key does indeed belong to Alice – hence a certificate system (e.g., a PKI – public key infrastructure) for validating public keys is needed. IBE obliviates the need for such a certificate system. In addition, IBE enables what we call *asynchronous* secret communication that is crucial to our construction of secure binding services: it allows one to establish a *forward* secret communication channel (using IBE for encryption) from a sender to a target identity, where the sender does not need to have prior knowledge or contact with the corresponding receiver; the private key corresponding to the target only needs to be generated by the PKG on-demand when the receiver requests for it. In contrast, in the conventional public key systems such as RSA, a public-private key pair must be pre-generated for each receiver before communications can proceed. For a detailed description of IBE, we refer interested readers to [4].

B. Secure Binding using DHTs: Why IBE?

Our goal is to build a *generic* secure binding service as a *trusted* infrastructure service, using which users can deposit bindings (e.g., email address and location mappings etc.), and query for bindings. As an infrastructure service, the service must be scalable and robust. To this end, we employ DHT techniques for building such a service. Note that our system is *not* a "peer-to-peer" system in the sense that it is not built using end hosts of users. In fact, nodes constituting the system are part of an infrastructure service: only authorized nodes can join the system. In this paper Chord is used as our choice of DHT but our approach can be easily extended to other DHTs.

The basic operations of a binding service using DHT are very similar to those of any DHT-based look-up services: a user u inserts (or "puts") a binding (a $\langle$`key, value`$\rangle$ pair) into the service by hashing `key` to an id in the flat id space, id_{key}, and sends the binding to the service using id_{key} as the target (i.e., destination). For a querier who wants to query for this binding, it simply sends a query message to the service using id_{key} as the target and the corresponding value is returned. To ensure the trustworthiness and basic security of the service, we must guarantee the following: i) only the owner of the binding, u, can update or delete the binding; ii) the returned binding to queriers is indeed from u. In particular, a secure binding service must be robust under *the "man-in-the-middle" binding poisoning attack* model, namely, an attacker interposed between users and the service cannot alter binding insertion requests from owners to the service or binding query replies from the service to the queriers.

One may argue that such service can be easily implemented using traditional public key crypto-systems such as RSA or other techniques. For example, OpenDHT [5] employs public key crypto mechanisms for authenticated `put/get` operations[2]. Unfortunately these authenticated `put/get` operations

[2]Under the authenticated put/get mode, each owner has a public/private key pair, denoted K_P and K_S, respectively. To insert a binding of key-value (k, v), an owner sends the following to the service: k, v, K_P, a nonce n , an expiration time t, and $\sigma = \{H(k, v, n, t)\}_{K_S}$, where $\{X\}_{K_S}$ denotes the digital signing of X with K_S and H is a secure hash function (e.g., SHA-1). To retrieve the binding, a querier sends the following $\{k, H(K_P)\}$ to the service and the service returns $\{(v, n, t, \sigma)\}$.

are not robust against the "man-in-the-middle" binding poisoning attacks, as the attacker may intercept the `put` message sent by a binding owner, replaces the public key with its own, and re-signs the message, and sends it to the service, which has no way to verify the bogus `put` message[3]. Moreover, in order for a querier to look up a binding using a binding key, it must know *a priori* the public key associated with the binding key, *which in itself requires a secure binding service*.

The public key infrastructure (PKI) can be viewed, in a sense, as a secure binding service that binds a public key to its owner with a certificate signed by a trusted third party, a *certificate authority* (CA). Hence a PKI-based approach can be used to build secure binding services (e.g., as used in DNSSEC and CoDoNS [7]), *provided that the owner's identity is part of the binding key.* In this case, to deposit a binding, an owner simply inserts a signed binding (using its private key) together with its certificate into the binding service. The advantage of such an approach is that the binding service becomes nothing but a repository of the signed bindings, no additional security mechanism needs to be provisioned. However, such a PKI-based approach is *not* robust against the "man-in-the-middle" binding poisoning attacks, when the (certified) owner identity is *not* part of the binding key – a necessary requirement for a *generic* binding service. To see why this approach fails to satisfy the two security requirements posed earlier, consider an "imposter" who also has an RSA key pair certified by the CA, and interposes itself in between users and the binding service. This attacker can intercept either the binding insert message from a binding owner or the query reply returned by the service, replace the original signed binding to a bogus one signed using its own public key, and change the attached certificate with its own. Neither the binding service nor users have any way to detect that the received binding has been tampered and is bogus. To circumvent this problem, a user has to generate an RSA key pair for each binding key (or sub-key) string that it might want to bind, and asks the CA to certify the corresponding public key. This requires either the user to anticipate all such bindings and have the CA to issue certificates beforehand, or the CA to be available at the time such a certificate is needed. In either case, it is not very scalable, in particular, in a dynamic environment where bindings are generated frequently and on-demand.

An alternative solution is to create secure channels between users and the service such that all operations are carried out over secure channels. Such an approach is robust against "man-in-the-middle" attacks as attackers can no longer intercept messages and inject bogus information at will. However, it is particularly challenging to establish secure channels in a DHT-based system using traditional public key crypto-mechanisms. For example, we can use multiple RSA public-private key pairs, each for one portion of the id space and store private keys at appropriate nodes. When a user wants to send a message to the system destined for some id, it simply encrypts the message using the right RSA pubic key and the node responsible for its message would decrypt the message using the corresponding RSA private key. However, this requires users to know which public keys to use for which portion of the id space; when the number of such public-private key pairs is large, it needs a secure binding service in itself to manage such public key to (portion of) id space mappings. In the extreme case, for example, a public-private key pair a priori must be generated for each id and distributed. With an id space of, say, 160 bits, this is clearly not scalable.

We now illustrate how one can build a *generic* secure binding service using IBE that does *not* rely on PKIs for security. In addition, our approach is robust against the "man-in-the-middle" binding poisoning attacks. The basic ideas are as follows. For an owner u who wants to insert a binding (a $\langle$`key, value`$\rangle$ pair), it employs IBE to encrypt the binding, together with a *secret symmetric key* (for a pre-specified *symmetric* encryption scheme such as AES) and a *nonce* (e.g., a random number), using the hashed id id_{key}, and sends the encrypted binding insert message to the binding service with the target id_{key}. The "root node" of id_{key} retrieves the private key corresponding to id_{key} from the PKG and decrypts it. If the insertion is successful, it returns a confirmation message containing the nonce encrypted with the secret symmetric key. Otherwise, a failure message is generated. From the confirmation message, the user can verify that its binding is indeed inserted into the binding service successfully. Likewise, for a user to look up this binding, it simply generates a query message together with a secret symmetric key and a nonce encrypted using id_{key}, and sends the encrypted binding query message to the binding service with id_{key} as the target. The binding service returns a reply message containing the binding and the nonce encrypted using the symmetric secret key. By successfully decrypting the reply message and verifying the nonce, the querier can be assured that the returned binding is indeed authentic. Any imposter interposed between the users and the binding service will not be able to tamper or inject bogus bindings into the service, nor return such to queriers.

C. Discussion and Other Related Work

Namespace management is a key *policy* issue in any binding service: who owns a name (or key) and has the right to bind the name (key) to a value? In this paper we attempt to separate such policy issues from the basic mechanisms for building secure, scalable and robust binding services, while *at the same time* providing the necessary "plug-ins" or interfaces to the namespace and policy management modules. In other words, these policy management modules, albeit an indispensable part of any binding service, are made "exogenous" to the basic operations of Vault. This is accomplished through a

[3]We note here that the SFRtags used in Semantic Free Referencing (SFR) [6] (also the immutable `put/get` operations of the OpenDHT), on the other hand, is robust against the "man-in-the-middle" attacks. However, due to the use of the "self-certifying" key technique (where the SFRtag is a secure hash of the value (e.g., URI) to be looked up and and the public key of the owner), SFR (also OpenDHT with immutable `put/get` operations) is *not* a binding service in its usual sense, as in a binding service a binding key (e.g., a phone number) may be bound to any legitimate binding value (e.g., IP address) and vice versa. Furthermore, SFR still requires a secure binding service to bind the public key to the owner of a value to be looked up.

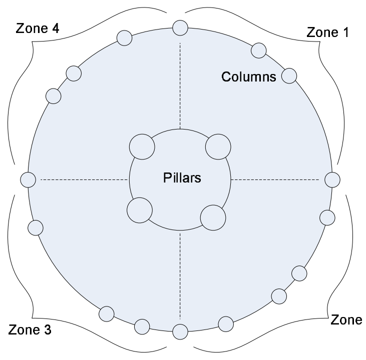


Fig. 1. Illustration of Vault architecture with four pillars and twelve columns.

required user registration process and separating the first-time binding insert operation from subsequent binding update operations so that user credential verification, namespace (key) right management, and other policy issues can be enforced by invoking the appropriate policy management modules. These points will be further discussed in Section III.

Identity-based cryptography has been an active research area in cryptology [4], [8]. IBE has also been used in several networked applications and systems such as IBE-based email systems [4] and secure opportunistic communications in disconnected networks [9]. Secure-i3 [10] uses constrained triggers based on secure hash mechanisms to prevent attackers from launching attacks against the system. To the best of our knowledge, our system is the first to combine IBE and DHTs to offer a *generic* secure binding service without the need of a trusted third-party service such as PKI.

III. Vault: Architecture and Operations

In this section we present the overall architecture and operations of Vault. Vault is built using a two-level architecture that enables efficient key management as well as better control over the service. At the core of the system is a special set of nodes called *pillars* that perform critical functions to ensure the security and proper operations of the system. In particular, each serves as a PKG in IBE. We assume that pillars are highly fault-tolerant and are always available. Pillars do not directly interact with users of the system, and do not store any user bindings. User binding management, storage and queries are handled by remaining nodes – called *columns* – which form an outer ring (the column ring) using Chord. As part of an infrastructure service, columns are assumed to be trusted; but unlike pillars, they may be dynamically added to or removed from the system. Fig. 1 shows an example of a Vault architecture. Pillars form a logical (inner) ring among themselves; they also act as logical separators for the column ring, partitioning it into several *zones* of equal size. Each pillar manages one zone, generating and issuing the private keys for *id*'s that fall within its zone. For instance, suppose we have an n-bit id space, and 2^m pillars. Then the column ring is divided into 2^m zones of size 2^{n-m}: for $i = 0, \ldots, 2^m - 1$, the ith zone contains the portion of the id space $Z_i = [i \cdot 2^{n-m}, (i+1) \cdot 2^{n-m})$ managed by the ith pillar P_i. To bootstrap Vault, for simplicity we assume

$root(id_k)$	A root column of id_k
s_k	Symmetric Key
n_u	Nonce from the user
$E_{s_k}(M)$	Symmetric encryption of M using s_k
$IBE^E_{id_k}(M)$	IBE encryption of M using id_k
$IBE^D_p(C)$	IBE decryption of C using p

TABLE I
Notations

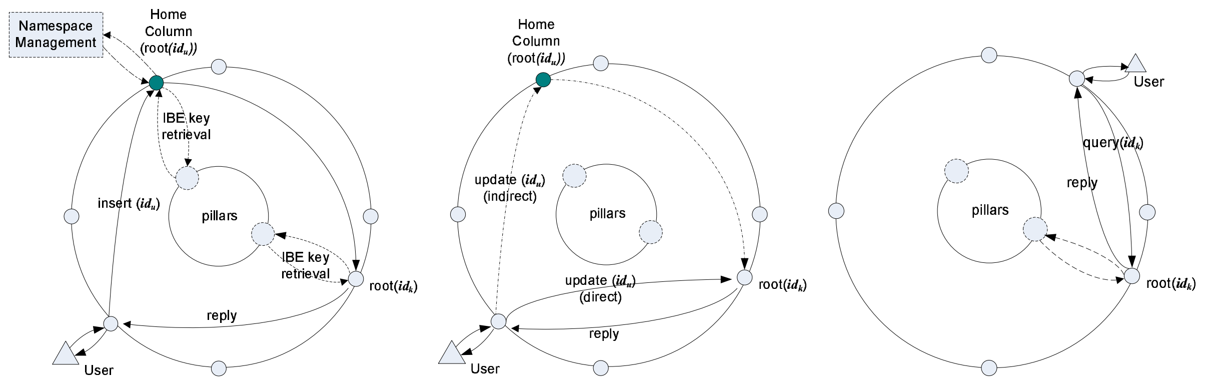
that we have as many columns ($\geq 2^m$) as pillars, where 2^m of the columns are placed at one end of the 2^m zones (i.e., they have pre-assigned id's, $i \cdot 2^m$, for $i = 0, \ldots, 2^m - 1$), as "anchor" columns. Although this is not necessary, it makes our exposition easier.

A. User Operations

Here we describe user operations in Vault. Table I lists some notations we used. Vault is used by two types of users: *owners* of bindings and *queriers* who look up bindings stored in Vault. This distinction is important to enforce security and accountability in our system and to facilitate namespace and other policy management. Before an owner can insert a binding into Vault, we require it to register with the system. The registration process serves several purposes: 1) to assign the user a unique user id, id_u, in the id space, and thereby also a (logical) *home column*, i.e., $root(id_u)$; 2) to establish an initial binding between id_u and certain user credentials such as user name, password, organization etc., and store such binding at the user's home column; and 3) to provide a natural interface with the namespace and other policy management systems (which are outside of Vault).

Once an owner u has obtained an id_u and stored its credential *cred*$_u$ at its home column, it can insert, and subsequently update or delete, bindings into Vault. To enable namespace management and policy enforcement, we require that binding insert operation always go through a user's home column, which checks the user's credentials and interacts with appropriate namespace management systems to ensure that the user has the right to insert the requested binding. To insert a new binding B of key-value, (k, v), the user u with id_u and *cred*$_u$ generates an insert request $m_I = IBE^E_{id_u}(id_u, cred_u, B, s_k, n_u)$. The user sends m_I to Vault with id_u as the target. Upon receiving the request, the home column ($root(id_u)$) processes it using Alg. 1 and forwards the request to the target $id_k = H(k)$, where H is a secure hash function. The request is then processed at $root(id_k)$ using Alg. 2. Either a confirmation response containing n_u or a failure response encrypted using the secret symmetric key s_k provided by the user is returned. Fig. 2(a) shows the steps taken for a binding insert request.

Once a binding has been successfully inserted, subsequent binding update or deletion can be carried out using two modes: direct and indirect. In the direct mode, the user encrypts a binding update $B' = (k, v')$ using the secret symmetric key s_k established with Vault earlier, and sends the encrypted update request, $m_U = E_{s_k}(B', n'_u)$, to Vault with id_k as the target. Upon receiving m_U, $root(id_k)$ looks up the symmetric key



(a) Binding Insert (b) Binding Update (Direct versus Indirect) (c) Binding Query

Fig. 2. Illustration of user operations

Algorithm 1 `INSERT` at Home Column: $root(id_u)$

```
 1: on receiving a binding insertion $(id_u, m_I)$;
 2: $pkey := get\_key\_from\_cache(id_u)$;
 3: if (!pkey) then
 4:     $pkey := get\_key\_from\_pillar(id_u)$;
 5: end if
 6: $(id_u, cred_u, B, s_k, n_u) := IBE^D_{pkey}(m_I)$;
 7: if $cred_u$ matches user credential then
 8:     consult namespace management policy to verify the ownership of $B$;
 9:     if verification = SUCCESS then
10:         $id_k := H[B.k]$;
11:         $m_I := IBE^E_{id_k}(id_u, B, s_k, n_u)$;
12:         $sig_{root(id_u)} := gen\_signature(m_I)$;
13:         send $(id_k, m_I, sig_{root(id_u)})$ to $root(id_k)$;
14:     end if
15: end if
```

Algorithm 2 `INSERT` at Column: $root(id_k)$

```
 1: on receiving a binding insertion $(id_k, m_I, sig_{root(id_u)})$;
 2: verify $sig_{root(id_u)}$;
 3: $pkey := get\_key\_from\_cache(id_k)$;
 4: if (!pkey) then
 5:     $pkey := get\_key\_from\_pillar(id_k)$;
 6: end if
 7: $(id_u, B, s_k, n_u) := IBE^D_{pkey}(m_I)$;
 8: if $B.k$ exists in the binding table then
 9:     return $r_I := E_{s_k}(\texttt{BINDING\_COLLISION}, n_u)$;
10: end if
11: Store $id_k$, $id_u$, $B$, $s_k$;
12: Set timer for $s_k$;
13: return $r_I := E_{s_k}(\texttt{INSERT\_SUCCESS}, n_u)$;
```

using id_k, and decrypts m_U, and updates the binding. Indirect update mode is used if the symmetric key s_k is expired and works in a similar manner as the binding insert operation: the request is sent to id_u first before being forwarded to id_k. Fig. 2(b) shows the steps for binding update in the direct and the indirect modes. The binding delete operation works in exactly the same way as the binding update operation, except that the binding is deleted by $root(id_k)$. Since no IBE operation is invoked, update and delete in the direct mode is far more efficient, thus it is particularly suitable in a dynamic environment where frequent updates are required.

Queriers are in general *not* required to register with Vault. In other words, any user is allowed to query Vault for bindings. To query a binding B with a binding key k, a querier sends a query request containing k, a secret symmetric key s_k, and a random nonce n_u encrypted with id_k to Vault with id_k as the target id. When $root(id_k)$ receives the request, it decrypts the request (upon obtaining a private key from the pillar) and checks if it has a binding B with a key k stored. If B exists, $root(id_k)$ encrypts B and n_u using s_k and sends the reply back to the user. If not, a failure message is generated by encrypting a `BINDING_NOT_FOUND` response and n_u with s_k. Fig. 2(c) shows the steps in the binding query operations. Because IBE operations are computationally expensive, as a general design principle, we only use IBE for the *first time* id-based communication; symmetric encryption is always used for the *reverse* or *subsequent* communications by including a secret symmetric key in the first-time forward IBE channel. The use of secret symmetric keys prevents attackers from returning any bogus messages and the (random) nonces included in user requests allows the user to distinguish between messages and to defer replay attacks.

B. Internal Operations

We now describe the internal operations of Vault to ensure the security of the system. Since Vault is an infrastructure service, only authorized nodes (columns) which have pre-assigned id's can join Vault. A column with a pre-assigned id is also given the private key corresponding to its id, which is used to generate a signature for identity authentication and to sign its messages using an *identity-based signature* (IBS) scheme [8]. Using IBS, other columns in Vault can verify the authenticity of its id before allowing it to join Vault. For a column to join Vault, we assume it knows at least one existing column currently active in Vault for bootstrapping purpose.

Having a pre-assigned id and proving its authenticity, however, is not sufficient in warranting a column to retrieve private keys from pillars. A key design challenge in Vault is to ensure that a column must be able to prove to a pillar that it is *currently active* in Vault, and is indeed *responsible for the id space range* it claims. To address this challenge, we devise an innovative mechanism – *secure bond* – to enable neighboring columns to securely bond to each other and lock in the portion of the id space each owns. This secure bond between neighboring columns is created by using *two-way hash chains* consisting of two secure hash chains, one in each direction. Recall that a hash chain is a crypto-primitive in

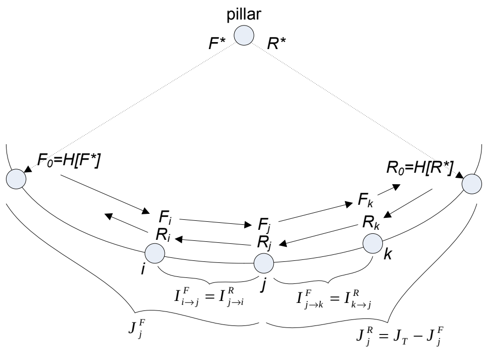


Fig. 3. Secure Bond using two-way hash chains.

which a root value r_0 can be used to derive all later values r_i for $i > 0$ by repeatedly applying H (e.g., SHA) to r_0. A value r_i in the chain can be used to derive all later values r_j for $j > i$, i.e., $r_j = H^{j-i}[r_i]$ where $j - i$ is the number of times we apply H to r_i in order to obtain r_j. However, r_i cannot be used to generate earlier values in the chain since inverting the secure hash function is computationally infeasible. We refer to j as the (absolute) jump of r_j (from r_0) and the value $j - i$ the *relative jump* from r_i to r_j. Given r_i, r_j for $j > i$ and the relative jump $j - i$ between them, we can use one value to verify the other by computing $r_{temp} = H^{j-i}[r_i]$ and compare r_{temp} with r_j.

We now show how we use the two-way hash chains to construct the secure bond between neighboring columns. To better illustrate this concept, we give an example using three columns i, j, and k located in the same zone such that i is j's predecessor, and k is j's successor. We assume each zone has a fixed total jump J_T that is proportional to the logarithmic of the size of the id space each zone has[4]. As shown in Fig. 3, the pillar issues root values $F_0 = H[F^*]$ and $R_0 = H[R^*]$ to two columns at two ends of its zone, where F^* and R^* are two secret values known only to the pillar. These two columns then pass some secret information derived from F_0 and R_0 in the forward and the reverse direction[5], and create two hash chains in the zone, the F-chain (forward) and the R-chain (reverse), respectively. We first establish the F-chain before we establish R-chain. When a column j receives a forward zone secret (ζ^F_j) from its predecessor (column i), it computes a relative jump $I^F_{j\to k}$ to its successor k. Column j then generates a new ζ^F_k using $I^F_{j\to k}$, and passes it securely to the successor k. This process is then repeated at the successor, and so on. Once the F-chain is established, we pass reverse zone secrets ζ^R from successors to predecessors using the relative jump I^R(=I^F, the forward relative jump) to establish the R-chain. Alg. 3 gives the details of the computation and passing of forward and reverse zone secrets from column j to its successor k and its predecessor i, respectively. The forward zone secret ζ^F_j of the column j contains a tuple $(F_j, I^F_{i\to j}, J^F_j)$ where $F_j = H^{I^F_{i\to j}}[F_i]$, $I^F_{i\to j}$ is the relative jump chosen by i to its successor j, and J^F_j is the absolute jump of j in the forward direction. We have that $F_j = H^{J^F_j}[F_0] = H^{J^F_j+1}[F^*]$. Similarly, j's reverse zone secret ζ^R_j contains a tuple $(R_j, I^R_{k\to j}, J^R_j)$, where $J^R_j = J_T - J^F_j$. We note that the forward and reverse relative jumps between two neighboring columns are the same in each direction, i.e., $I^F_{i\to j} = I^R_{j\to i}$, and the sum of all relative jumps equals to J_T.

[4]Hence the hash chain can be computed in polynomial time in n, the number of bits of the id space.

[5]*forward* means counterclockwise and *reverse* means clockwise, we use superscript F and R to denote these two directions

Algorithm 3 Secure Bond using Two-Way Hash Chains

```
 1: //secure bond between $j$, its pred. $i$, and its succ. $k$;
 2: $\zeta^F_j = (F_j, I^F_{i\to j}, J^F_j)$;
 3: $\zeta^R_j = (R_j, I^R_{k\to j}, J^R_j)$;
 4: //compute $\zeta^F_k$, $\zeta^R_i$, line 5–13
 5: $I^F_{j\to k} := o(\log(|id_j - id_k|))$; //logarithmic jump with some scaling factor
 6: $I^R_{j\to i} := I^F_{i\to j}$;
 7: $F_k := H^{I^F_{j\to k}}[F_j]$;
 8: $R_i := H^{I^R_{j\to i}}[R_j]$;
 9: $J^F_k := J^F_j + I^F_{j\to k}$;
10: $J^R_i := J^R_j + I^R_{j\to i}$;
11: send $\zeta^F_k := (F_k, I^F_{j\to k}, J^F_k)$ securely to $k$;
12: send $\zeta^R_i := (R_i, I^R_{j\to i}, J^R_i)$ securely to $i$;
13: store $I^F_{j\to k}$;
```

Given F_j, R_j, J^F_j and the total jump J_T, we can pin down the position of the column j (relative to other columns) in the zone. To see why this is the case, suppose that the column j wants to move its position in the zone. If j wants to move in the forward direction, it can do so by creating a new value F'_j such that $F'_j \succ F_j$ in the F-chain[6]. This means column j must increase its absolute jump J^F_j to a larger value $J^{F\prime}_j > J^F_j$ so that $F'_j = H^{J^{F\prime}_j+1}[F^*]$. Since the total jump in a zone is fixed at J_T and $J^R_j = J_T - J^F_j$, an increase in J^F_j means a corresponding decrease in J^R_j. This requires column j to obtain a smaller value of $R'_j \prec R_j$ in order to "move" successfully in the forward direction. However, this is not possible due to the properties of the secure hash chain. Similarly, j cannot move in the reverse direction as it cannot obtain a smaller value $F'_j \prec F_j$. Thus, each column's F, R, and J values enforce each other and *uniquely determines a column's position (relative to other columns) in the zone*. In addition, a column cannot fake the position of its predecessors and successors.

However, simply pinning down a column's relative position in the zone does not tell us the portion of the id space it owns. We utilize j's predecessor i to obtain this information as each column's id space range is determined by its own id and its predecessor's id. To do so, the predecessor (column i) passes a *timed* token T_j encrypted using pillar's id id_p to column j. The token includes the following information: F_i, R_i, J^F_i, id_i, and a timestamp t_s, i.e., $T_j = IBE^E_{id_p}(F_i, R_i, J^F_i, id_i, t_s)$. (The signature of column i is included as part of its id). The token is periodically refreshed with a new timestamp t_s. The pillar can then use this information to authenticate and verify a column before issuing private keys. The token is also regenerated and passed to the new successor when a predecessor has detected that an old successor has left the system and/or a new successor has been established.

[6]Here $b \succ a$ means that b has a bigger (absolute) jump than a

When a column j wants to retrieve a private key for some id_{key} from the pillar, it sends a key request message m_R by disclosing the following: the forward and reverse zone secrets F_j and R_j, the jump J_j^F and $I_{i \to j}^F$, and the timed token T_j as well as its id own id_j (with its signature) to the pillar securely. Key request operations are carried out over secure channels as a user would do for binding operations (thus protecting the operation against man-in-the-middle attacks): the column j encrypts the above information together with a symmetric key and a nonce with the pillar id id_p; the pillar returns the private key and the nonce back using the symmetric key. As this operation When the pillar receives the key request, it checks the timestamp in the token to determine if the column j is active in Vault at the moment, and obtains its predecessor's id. Together with the column j's own id, the pillar can determine the exact portion of the id space the column j owns, and determine whether the column j is indeed responsible for the requested key id id_{key} (see Alg. 4). (Note that the id's of both columns i and j can be verified by the pillar using their respective signatures.) Since the token from the column j's predecessor is periodically refreshed, after the column j has left Vault after some time, it can no longer retrieve private keys from the pillar.

Algorithm 4 Key Retrieval at Pillar

```
 1: on receiving key request $id_j, m_R$ for $id_{key}$ from $j$;
 2: $pkey := gen\_private\_key(id_p)$; //can be stored for later use
 3: $(F_j, R_j, T_j, J_j^F, I_{i \to j}^F) := IBE_{pkey}^D(m_R)$;
 4: $(F_i, R_i, J_i^F, id_i, t_s) := IBE_{pkey}^D(T_j)$;
 5: if $t_s$ is still valid then
 6:   if $F_i = H^{J_i^F + 1}[F^*]$ and $R_i = H^{J_T - J_i^F + 1}[R^*]$ then
 7:     if $F_j = H^{I_{i \to j}^F}[F_i]$ and $R_i = H^{I_{i \to j}^F}[R_j]$ then
 8:       if $id_{key} \in (id_i, id_j)$ then
 9:         $pkey_j = gen\_private\_key(id_{key})$;
10:         return $pkey_j$ securely to $j$;
11:       end if
12:     end if
13:   end if
14: end if
```

Once the initial two-way hash chains have been established, F and R values for each column remain fixed and do not need to be re-calculated regardless of column dynamics (join/leave). When a new column h joins the system between columns j and k, it first requests a forward zone secrets from its predecessor j. The column j computes a new relative jump $I_{j \to h}^F (< I_{j \to k}^F)$, the corresponding ζ_h^F, and the relative jump $I_{h \to k}^F$ from h to k for h. The column h then computes a new ζ_k^F for h's successor k based on $I_{h \to k}^F$. By maintaining the relative jump $I_{j \to k}^F = I_{j \to h}^F + I_{h \to k}^F$, the column k's F_k value remains constant. On receiving ζ_k^F, k computes ζ_h^R using the new relative jump $I_{k \to h}^R$ (=$I_{h \to k}^F$) and so on. When a column leaves the system, its predecessor and successor contact each other and calculate the relative jump between them so that consistency is maintained. For example, if j leaves the network, h now becomes the direct successor of i, and the relative jump $I_{i \to h}^F$ between i and h should be the sum of $I_{i \to j}^F$ and $I_{j \to h}^F$.

Operations	Overhead
`symmetric_encrypt`	41 μs
`symmetric_decrypt`	26 μs
`IBE_key_gen`	22 ms
`IBE_encrypt`	32 ms
`IBE_decrypt`	25 ms

TABLE II

COMPUTATIONAL OVERHEAD FOR CRYPTO-PRIMITIVES.

C. Further Enhancements

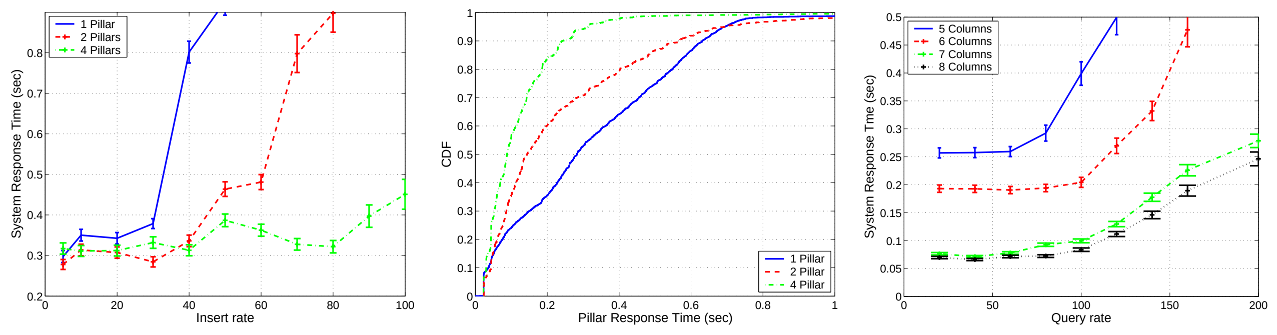
We briefly discuss several enhancements to Vault that can further improve Vault's performance and offer additional security. Vault inherits its scalability and robustness from DHT. However, as we will demonstrate in Section IV, due to the computational overhead of IBE and relatively large network latency, the overall performance of Vault needs to be improved, in particular, to handle sudden surges in request loads such as "flash crowds". We address the latency issue by introducing the concept of *local (Vault) proxies* and *query delegation*. A local proxy is a (trusted) column in Vault is located in close proximity to a user (or a group of users) that can perform query requests on behalf of users and cache returned bindings (similar to that of the local DNS), thereby considerably reducing the network latency for subsequent queries on the cached bindings. We use *binding delegation* to handle flash crowds. Binding delegation enables a column (say, $root(id_k)$) to replicate a binding it owns to a subset of columns it chooses. It also delegates certain private keys derived from the private key for the binding to these columns so that they can answer queries on behalf of $root(id_k)$. The binding replication and delegation can be done either in a pre-arranged fashion or on demand. Note that binding delegation allows multiple columns to answer queries for the same binding, thereby avoiding the single point of failure. Finally, although Vault may still be vulnerable to compromise (say, by insiders or due to other vulnerabilities outside the design and control of our system), given the built-in accountability and security in Vault, such attacks can plausibly be detected more easily with additional monitoring and defense capabilities for identifying inconsistency, misuses, and anomalies. Due to the space limitation, we refer interested readers to a technical report version of this paper [11] for more details on enhancements.

IV. EXPERIMENTAL EVALUATION

In this section we present evaluation results from a prototype implementation of Vault that has been deployed and tested on the PlanetLab as well as in our local network.

A. Local Experiments

We start with the experimental results we have obtained in a local testbed located in a local area network. These experiments allow us to evaluate the computational overheads of Vault operations, and how they are affected by the number of nodes (pillars and columns) as well as user request loads, while without accounting for the potential impact of (wide-area) network latency.



(a) Insert Response Time (b) Pillar Response Time (c) Query Response Time

Fig. 4. System Response Time in Local Experiments

	User	Column	Pillar
Insert	33 ms	57 ms	73 ms
(Direct) Update/Delete	37 μs	60 μs	N/A
Query	33 ms	25 ms	N/A

TABLE III
COMPUTATIONAL OVERHEAD FOR BINDING OPERATIONS.

We first measure the computational overhead of IBE crypto-primitives (as implemented in the Boneh-Franklin IBE library) and compare them with the symmetric crypto-primitives (using AES). We use the Time Stamp Counter (TSC) on a Pentium 4 2.4GHz Processor to count the number of clock cycles taken by each crypto-primitive and divide it by the CPU frequency to measure the computational overhead. The input to each primitive is similar to what is used in Vault operations. For example, we use 160-bit ids as inputs to measure the overhead of IBE private key generation. Table II shows the overhead of each crypto-primitive as the average of 30 runs. It is clear that IBE is far more expensive than the symmetric cryptosystem.

Next we measure the computational overhead of Vault operations. For this we use a simple set-up with three machines, one as pillar, one as column and one as an end host. For binding insertion, we use the user id id_u as the binding key (i.e., $id_k = id_u$). We also allow the column to cache the private key of the id once is retrieved from the pillar. Hence for binding update and delete in the direct mode, no IBE primitives are performed. Table III shows the computational overheads of each Vault operation as measured at user, column and pillar side. The results are obtained as the average of 30 runs for each operations. We omit the results for update/delete in indirect mode as they are similar to that of binding insert operations. The results show that the computational overhead for each binding operation depends largely on the number of IBE operations each entity needs to perform. The use of symmetric crypto-mechanism for binding update in direct mode significantly improves its performance.

We now illustrate how the number of pillars influence the performance of Vault operations. We use four machines as columns and vary the number of pillars from one to four. A number of other hosts are used to generate random binding insert requests at various rates. We measure the system response time, defined as the lapse from the time immediately after a user sends out a binding insert request to the time a reply from Vault is received. Fig. 4(a) shows the average system response (measured over a 15-minute duration) as a function of binding insert request rate when different number of pillars is used. The results show that as we increase the number of pillars, we improve the response time of Vault in handling binding insert requests. To better demonstrate this point, Fig. 4(b) shows the distribution for the pillar response time – the time immediately after a column sends out a key retrieve request until the time the private key is returned – at a request rate of 30 inserts/second. Similar patterns for the pillar response time distribution are observed for all other request rates. As we can see, by adding more pillars to the system we can reduce the overall load imposed on each pillar, thus yielding faster overall system response time.

To investigate the impact of the number of columns on the overall system performance, we use a setup with one pillar but vary the number of columns from 5 to 8 machines. We use the performance of query operation as a representative example to illustrate the results. To evaluate the query performance, we first insert 5000 random bindings to the system and generate queries for these bindings uniformly at different rates. Fig. 4(c) shows the result of query rate versus the (average) system response time (again over a 15-minute period), with 1 pillar and varying number of columns. Using only 5 columns, the average system response time starts to increase significantly once the query rate reaches 80 query/second. With 7 or 8 columns and only one pillar, the system can handle up to 120 query/second before we start seeing some increase in the average system response time. Furthermore, the average system response at 200 query/second is lower than that of 5 columns and 1 pillar at half the query rate, namely, 100 query/second. Similar observations hold for direct update and delete operations with results showing around 500 times improvement due to the use of symmetric crypto-mechanism.

Lastly we evaluate how binding delegation (see Section III-C) can improve the system performance, especially under "flash crowds" – when most users query for the same binding. For this experiment, we first set up a system with 1 pillar and 8 columns, with each of them serving some background queries at 5 query/second per column. We emulate the flash crowd effect by generating queries that target a specific binding at high rates. Without binding delegation, all these queries must be handled by one column. We see that at the rate of 35 query/second for the target binding, the average system response time quickly jumps beyond 500 ms. By replicating the binding to 7 other columns and thus serving queries for this binding at 8 columns instead of one, the system response time reduces to around 100 ms even if we increase the query rate for the target binding to 100 query/second. (Due to space limitation, we do not present the detailed plots here.) We see that binding delegation is an effective way to improve the overall system performance, in particular, under flash crowds.

From our local experiments, we show that without considering the potential impact of (wide-area) network latency, the computational overheads of Vault operations are dominated by IBE crypto-primitives, the price we pay for added security. However, by using "IBE-only-once" principle, we can employ symmetric encryption mechanisms for binding update/delete operations to significantly increase their performance. Furthermore, we can scale the system performance by increasing the number of columns and pillars. In addition, binding delegation can further improve the overall system performance, in particular, under flash crowds.

B. PlanetLab Experiments

Here we present the evaluation results from the PlanetLab deployment. We deploy Vault using from around 80 to 100 nodes on the PlanetLab as columns and pillars, and an additional number of nodes are also used as users of Vault to generate binding insert, update, delete and query requests. These nodes are geographically dispersed on the Internet.

Since PlanetLab nodes used in our experiments have varying processing powers and are heavily loaded with many experiments running concurrently at any given time, we first measure the time it takes for these PlanetLab nodes to execute IBE crypto-primitives. The measurement is done similar to our local testing by measuring CPU clock cycles. Table IV shows the results for several PlanetLab nodes with different CPU speeds, where each value is the average of three nodes at the same CPU speed at different time of the day for a total of 500 measurements. The standard deviation is given in the parentheses. Comparing with Table II, we see that the PlanetLab nodes perform much worse than local machines. In addition, faster machines does not necessarily give us better performance. The relatively large standard deviation also reflects the time variability in executing the crypto-primitives due to fluctuating CPU loads on the machines.

We now present some experimental results regarding the overall performance of Vault on PlanetLab. For these experiments, 75 PlanetLab nodes are used as columns and the number of pillars are varied from 4, 8, to 12. Additional PlanetLab nodes are used to emulate a large number of users of Vault to generate binding insert/update and query requests. For each experiment, a different request rate is generated by users from a certain distribution of binding keys. Before each set of experiments, we stop and restart Vault and allow a 15-minute initialization and stabilization period to establish zone secrets and ensure the system is stable and operates correctly.

mean (std)

CPU (GHz)	`IBE_key_gen` (ms)	`IBE_encrypt` (ms)	`IBE_decrypt` (ms)
1.8	57 (27)	83 (35)	58 (16)
2.4	188 (139)	241 (94)	202 (118)
2.8	102 (38)	133 (31)	106 (35)
3.0	125 (98)	155 (73)	127 (56)
3.06	118 (103)	155 (64)	114 (41)

TABLE IV
COMPUTATIONAL OVERHEAD IN PLANETLAB MACHINES

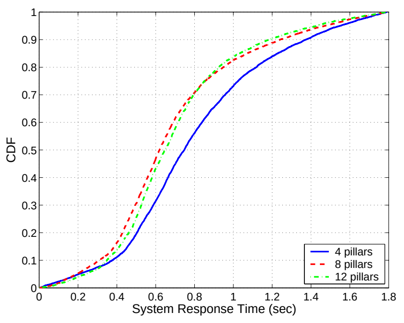
System - operation	Mean (ms)	$95^{th}\%$
Vault - Insert	725	733
Dummy V. - Insert	601	617
Vault - Query	494	497
Dummy V. - Query	346	347

TABLE V
SYSTEM RESPONSE TIME FOR VAULT AND DUMMY VAULT.

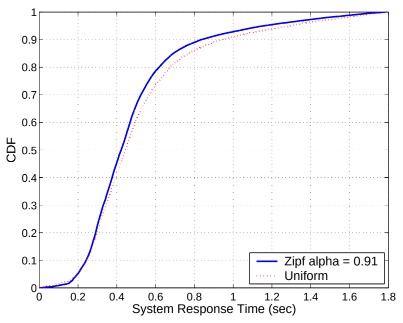
Each experiment then lasts about 45 minutes. Similar to the local testing, we measure the overall system performance from the users' perspective: the system response time is measured as lapse from the time immediately after a user request generated to the time a reply is received.

As a representative example, Fig. 5(a) shows the cumulative distribution function (CDF) of the system response time for binding insert operations that are generated uniformly at a rate of 15 insertion/second by users. We see that the median system response time (50% percentile) reduces from just below 0.8 seconds to 0.6 seconds when the number of pillars is increased from 4 to 8. However, there is almost no difference when the number of pillars is further increased from 8 to 12, suggesting the pillars are not the performance bottleneck. Fig. 5(b) shows the CDF of the system response time for query operations using the experimental setting of 75 columns and 8 pillars. The query requests (the binding keys) are generated from either a uniform or a Zipf distribution with $\alpha = 0.91$ (this value is chosen to to mimic typical DNS loads [7], [12]). From the figure we see that there is virtually no difference in the performance under these two distributions. In all cases, around 80% of queries are answered within 0.6 seconds and the mean query response time is around 500 ms. We have also conducted experiments using different user request rates and different number of columns (from 75 to 100). Due to space limitation, we do not present the results. The general observations are that under modest user request rates, increasing the number of pillars and columns improves the overall system performance initially; however after a certain threshold, further increasing the number of pillars or columns does not yield any significant performance improvement. Comparing these results with those of the local testing, it leads us to suspect that the wide-area network latency becomes a more dominant factor in determining the overall system performance.

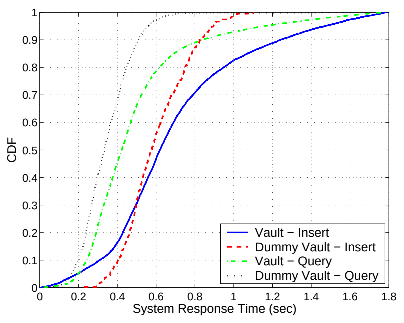
To investigate the wide-area network latency, in particular, Chord message routing latency, we implement a "stripped down" version of Vault – referred as the *Dummy Vault* – which mimics the message routing pattern of Vault but does not execute any binding and key retrieval operations. Hence the system response time of Dummy Vault includes mostly the



(a) Insert



(b) Query



(c) Vault and Dummy Vault

Fig. 5. System Response Time on the PlanetLab

wide-area network latency in routing messages. To accurately measure the network latency, we run Vault and Dummy Vault at the same time and send back-to-back binding requests to both systems. Fig. 5(c) and Table V compare the system response times of Vault and Dummy Vault (Dummy V. in Table V) for binding insert and query operations, where both systems contain 8 pillars and 75 columns. We see that Chord message routing latency indeed contributes a predominant portion (in general over 70%) of the overall system response time. Similar results also hold under other system settings.

Since the wide-area network latency (in particular, Chord routing latency) plays a dominant role in the overall system response, in the final set of experiments we investigate the performance benefits of query delegation via a *local Vault proxy*. For this, we use a PlanetLab node located on our campus network as the local Vault proxy. We set up several machines in our lab to emulate users and generate binding queries and measure the system response time for each request. With query delegation, queries are encrypted with the id of the local proxy instead of id_k for a given binding key, k. Local proxy would forward queries to Vault on behalf of users if they cannot be found in its local binding cache, otherwise it would reply with the cached bindings. Once a user has contacted the local proxy for the first time, a secret symmetric key can be used for subsequent query requests with no IBE operations. Our experimental results show that for bindings that are not found in the local cache, the system response time ranges from 800 ms to 1.2 seconds. For subsequent queries from new users (i.e., those that contact the local proxy for the first time) on the cached bindings, the average system response time is reduced to around 220 ms, with more than a four-fold reduction. The dominant factor in the response time here is the time for performing the IBE decryption at the local proxy. Furthermore, for queries on cached bindings from previous users (thus a secure symmetric channel has been established with the local proxy), the system response is only around 5 ms. This drastic improvement in performance is due to the much faster symmetric cyrpto-mechanisms. Note that because the local PlanetLab node that serves as our local proxy is fairly heavily loaded, crypto operations take much longer than the results we obtained in our local experiments (cf. Table II). Overall our results show that similar to DNS, we can significantly reduce the overall system response time by query delegation and caching.

V. Conclusion

In this paper we have developed a novel approach in building a *generic* secure binding system as a (trusted) infrastructure service by combining IBE and DHT. In building such a system – referred to as Vault, we have also developed several innovative mechanisms to address various important design and implementation issues. A prototype implementation of the system has been deployed and evaluated on the PlanetLab as well as in a local testbed. We believe our approach explores a new dimension in constructing generic secure binding systems and our system can be used as a fundamental building block to facilitate the development of next generation networks and network applications.

References

[1] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the internet. In *SIGCOMM '04*, New York, NY, USA, 2004. ACM Press.

[2] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Frans Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[3] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, London, UK, 2001. Springer-Verlag.

[4] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. *Lecture Notes in Computer Science*, 2139:213–229, 2001.

[5] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. Opendht: a public dht service and its uses. In *SIGCOMM '05*, pages 73–84, New York, NY, USA, 2005. ACM Press.

[6] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the web from dns. In *NSDI '04*, 2004.

[7] V. Ramasubramanian and E. Sirer. The design and implementation of a next generation name service for the internet. In *SIGCOMM '04*, New York, NY, USA, 2004. ACM Press.

[8] B. Libert and J. Quisquater. The exact security of an identity based signature and its applications. Cryptology ePrint Archive, Report 2004/102, 2004.

[9] A. Seth and S. Keshav. Practical security for disconnected nodes. In *Proc. First Workshop on Secure Network Protocols (NPSEC)*, November 2005.

[10] D. Adkins, K. Lakshminarayanan, A. Perrig, and I. Stoica. Towards a More Functional and Secure Network Infrastructure, UCB Technical Report No. UCB/CSD-03-1242, 2003.

[11] G.-H. Lu, C. Choi, and Z.-L. Zhang. Vault: A secure binding service. technical report. university of minnesota.

[12] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. In *IMW '01*, New York, NY, USA, 2001. ACM Press.