The CryptoID Key Management Protocols

Trevor Perrin
trevp@trevp.net

1. INTRODUCTION

The cryptoID system was introduced in [2]. That paper presented the cryptoID fingerprint and certificate formats. This paper presents the cryptoID key management protocols.

Section 2 briefly recaps the cryptoID approach. Section 3 recaps the cryptoID data formats. Section 4 presents the protocols.

2. THE CRYPTOID APPROACH

The cryptoID system is designed to support asymmetric cryptography in interactive inter-person communications such as voice-over-IP, text chat, and video-conferencing. The cryptoID system may also be used with technologies such as VPNs, wireless access points, the web, or email.

The cryptoID system is similar to PKI, PGP, or SPKI, in that it provides a framework for dealing with asymmetric keys. It differs in two chief respects.

2.1 Key Management vs. Key Distribution

First, the cryptoID architecture envisions a clear separation between private key management and public key distribution. We can explain this best by contrasting it with X.509 PKI. In PKI, the CA (or a network of CAs) handles both public key distribution and private key management. The CA issues the certificates used for key distribution. The CA also issues the CRLs used for key revocation, and issues new certificates when the subject wishes to change keys.

In the cryptoID system, private key management and public key distribution are separate, layered functions performed by different parties.

Private key management is the lower layer function, and should be performed by the subject, or by parties chosen by the subject and under her control. The task of private key management is for the subject to create a *crypto identity* for herself which is both resilient (in the face of key loss and compromise) and mobile (so she can use the same crypto identity on different devices and in different applications).

To accomplish this, the subject will choose some online trusted parties as her personal *certification authorities* (CAs) or *validation authorities* (VAs). CAs will certify new keys at the subject's request, enabling key replacement and mobility. VAs will revalidate keys periodically, enabling key revocation (by ceasing to revalidate). The subject will form her crypto identity by concatenating the public keys of these parties together, and hashing the result to create a fingerprint.

Public key distribution is a higher-layer function than private key management, in the sense that the output of private key management (a crypto identity) is the input to public key distribution. The task of public key distribution is for the relying party to learn the subject's crypto identity, probably by querying some third party he trusts. Thus while private key management centers around the subject and her trust decisions, public key distribution centers around the relying party and his trust decisions.

Now we see the advantage of separating these tasks: separation allows the subject and the relying party to make their trust decisions independently, without needing to coordinate, and without being constrained by the other party's choices. This flexibility is crucial in an anarchic environment like the Internet, where no globally-trusted authority exists. Separation also allows more specialized trusted third parties which focus on their particular tasks.

2.2 Public Key Distribution with Fingerprints

The second difference between the cryptoID system and PKI or PGP is that the cryptoID system handles all key distribution through fingerprint exchange, instead of certificates. The subject will form a fingerprint, aka a *cryptographic identifier* or *cryptoID*, as described above – she will hash a collection of *root keys* which she trusts to issue and validate her *key management certificates*.

These cryptoIDs will then be passed around through trusted directories, business cards, pen-and-paper, and other means. Fingerprints are preferable to certificates because they are easier for users to use and for developers to support:

- Fingerprints are easy for users because they are similar to things like phone numbers, mailing addresses, and email addresses which users are accustomed to exchanging and storing.
- Fingerprints are easy for developers because they don't require complex path discovery (like PKI) or a sophisticated trust-management GUI (like PGP). Applications only need an additional field in their address books where fingerprints can be stored.

3. DATA FORMATS

As we've, seen, the cryptoID system uses fingerprints for public key distribution, and certificates for private key management. This approach could be implemented with

the PKI, PGP, or SPKI fingerprint and certificate formats. However, these formats are less than ideal for this:

Conventionally, using fingerprints for key distribution has been looked down upon. As a result, little effort has been made to design easy-to-use, "ergonomic" fingerprints. In particular, current fingerprints are too long. CryptoID fingerprints are half the size of current ones, yet still offer an adequate security level.

Conventionally, certificates have been used for public key distribution, not private key management. As a result, more effort has gone into designing sophisticated naming, authorization, and policy languages than into providing key management functionality such as revalidation and threshold subjects.

CryptoID certificates don't need sophisticated naming or authorization languages since they are always issued from a crypto identity to itself. Instead, cryptoID certificates implement powerful SPKI-derived key management techniques such as *threshold subjects* and *timed revalidation.* CryptoID certificates accomplish these techniques in a more elegant fashion than SPKI with the simple yet flexible mechanism of *key expressions.*

3.1 The CryptoID Fingerprint Format

Current PKI and PGP fingerprints are hex-encoded SHA1 values. These fingerprints are 40 characters long and have a security level of 160 bits, which means an attacker would have to construct around 2^{160} fingerprints before discovering a collision with a victim's fingerprint. Such a collision would allow the attacker to usurp the victim's crypto identity.

CryptoID fingerprints are only 20 characters long. This makes them much more user-friendly: they're easier to read, write, compare, and memorize, and they're more likely to fit in cramped places like business cards, small LCD screens, and paper directories. Below are some example cryptoIDs:

`dq44t.z3ryz.o5ufw.zqcr3`

`friku.zgent.ofse4.jykti`

`cas93.b75d4.ex79k.usxrj`

`fa7v8.d7gwb.c53tw.5f3fu`

CryptoIDs have a parameterizable security level. Depending on how much computation is performed in creating it, a cryptoID's security level could be 104, 112, 120, 128, 136, 144, 152, or 160 bits. This is accomplished through a technique called "hash extension" [1], which is best understood as a form of fingerprint compression.

In essence, the creator of a cryptoID repeatedly places different *modifier* values in the to-be-hashed data and hashes it until a hash value is discovered that begins with a number of zero bytes. Such a hash value can be run-length-encoded into a more compact form. For example, if a hash value begins with three bytes worth of zeros, the cryptoID creator writes the number "3" into the first four bits of a cryptoID, then writes the next 96 bits of the hash value into the cryptoID, and base32-encodes the resultant 100 bits to get the final cryptoID.

This cryptoID can be "uncompressed" to yield a 120 bit hash value prefix (three bytes of zeros + 96 bits) with a 120 bit security level. To get a security level this high, a cryptoID creator has to search through, on average, around 2^{24} modifiers. The author's Pentium 4 1.7 GHz laptop can search over 900,000 modifiers per second. This yields the following average search times for particular security levels:

112 bits: $1/20^{th}$ of a second

120 bits: 13 seconds

128 bits: 56 minutes

Given current technology, these values can be viewed as low, medium, and high security levels. As processors get faster, people will move to more secure cryptoIDs, so cryptoIDs have substantial "future-proofing" built in.

If you're happy with a lower-security cryptoID, it's easy to generate a few thousand of them and skim through looking for a nice, easy-to-memorize one. For example, the $1{,}614^{th}$ 112-bit cryptoID the author generated in such a trial was:

`dubo9.sanre.wivip.wqsqy`

For more details on the cryptoID fingerprint format, see section 3.3 in [2].

3.2 The CryptoID Certificate Format

When two parties are negotiating secure communications, they will exchange cryptoID certificate chains. A cryptoID certificate chain is an XML element that establishes that the end-entity key (the last key in the chain) is certified under a particular cryptoID, for a particular set of protocols, for a particular period of time. By restricting the end-entity key to a set of protocols for a period of time, we limit the damage done if it is compromised.

The certificate chain always contains a root certificate, which holds the <*modifier*> element. The certificate chain may contain certificates previous to the root which can be shared by many cryptoIDs. The certificate chain may also contain certificates after the root, which are specific to a particular cryptoID. The root certificate and all previous certificates are hashed to create the cryptoID.

Each certificate may contain an expiration time and a list of authorized protocols. Each certificate also contains a *certID*, which is used to refer to the certificate in key management protocols. You might expect each certificate to contain a single public key and a signature over the certificate's contents by the preceding certificate's key. Instead, cryptoID certificate chains contain *lists* of public keys and signatures alongside the certificates.

Example <certChain>

```
<certChain chainID="tz2lR4taBk4rgCv9eITftOnQogo=" cryptoID="fsaxu.3cnqn.99kwa.ju93y"
           xmlns="http://trevp.net/cryptoID">
    <certs>
        <cert certID="Bpd06cPH1JcT4yuyue7wTV7u21c=">
            <keyExpression expr="((2 of A,B,C) and (D or E))">
                <keyHash>K8FcXZQvWZUgZdGgnmfZq17qeVM=</keyHash>
                <keyHash>xfFTqThoskJ7vPknGvlndXlxGNQ=</keyHash>
                <keyHash>Jb3e71i+IshcrnQxGElxMiBIT44=</keyHash>
                <keyHash>mOBrWJxjPzOAxuNzVtXfMu8HvXs=</keyHash>
                <keyHash>qwjAyFzjmgfacBDrs7lPOq16ids=</keyHash>
            </keyExpression>
            <modifier zeroCount="2">8701372</modifier>
        </cert>
        <cert certID="Md3BOL7J3Cpg5a5+BcZqJZKgG1Y=">
            <keyExpression expr="(F and (D or E))">
                <keyHash>FjhqXDC+MTkhP8TjUO0AduGqNOo=</keyHash>
            </keyExpression>
            <protocols>
                <protocol>http://trevp.net/instant-messaging</protocol>
            </protocols>
        </cert>
    </certs>
    <publicKeys keys="ABDF">
        <publicKey xmlns="http://trevp.net/rsa">
            <n>vSm/WK4D9vZWuaRb5PtZ5WJL3phj2PfU+BKk1qT...</n>
            <e>Aw==</e>
        </publicKey>
        <publicKey xmlns="http://trevp.net/rsa">
            <n>zgD6QBJ+f9jXdhqbVUr6b7UenAdwDVX58acwdLF...</n>
            <e>Aw==</e>
        </publicKey>
        <publicKey xmlns="http://trevp.net/rsa">
            <n>uerDiuvPz81wTt0T1RYpW9eQqyJP4h6BvxTONvf...</n>
            <e>Aw==</e>
        </publicKey>
        <publicKey xmlns="http://trevp.net/rsa">
            <n>oGJvKtYMYyFZ9UJumfrQOF31efCnzgA1joTq1d5...</n>
            <e>Aw==</e>
        </publicKey>
    </publicKeys>
    <signatures>
        <signature chaff="l+u3m4YqZFFBL8LW1BjFwS" key="A" listCA="0"
                   notAfter="2005-05-23T23:56:48Z" sigAlgo="pkcs1-sha1">eBk8pg...</signature>
        <signature chaff="XVOqVCydy95kihYL1jEHSi" key="B" listCA="0"
                   notAfter="2006-05-06T05:17:11Z" sigAlgo="pkcs1-sha1">mgd3eA...</signature>
        <signature chaff="jSOaSL90RYyGpmsiV6lb9R" key="D" listVA="01"
                   notAfter="2004-09-25T23:59:40Z" sigAlgo="pkcs1-sha1">hjyYhr...</signature>
    </signatures>
</certChain>
```

3.2.1 Public Keys

Instead of containing a public key, each certificate contains some *key variable bindings* and a *key expression*. A key variable binding associates a *key hash* with a variable name. The first binding in the chain binds the variable "A", the next "B", and so on to a maximum of "Z". These variables are associated with SHA1 hashes of public keys.

The key expression specifies which keys can exercise the power vested in the certificate. For example, "(A and B)" means keys A and B must both act in agreement, and "(E or (2 of B,H,C))" means key E can act alone, or else some pair of the remaining keys needs to act in concert. A key expression may use key variables bound in the current certificate or in any previous one. Keys exercise their power by issuing signatures over the certificates.

The above certificate chain contains two certificates, four public keys, and three signatures. Every key that issues a signature in the signatures list will have its public key present in the public keys list, so that the relying party can verify the signatures. In this case, keys A, B, and D have issued signatures, so these keys are present. In addition, key F is the first key mentioned in the last certificate, which makes it the end-entity key, so it is included in the public keys list as well.

Why are public keys factored out of the certificates and listed separately? For two reasons: First, keys that aren't needed to verify signatures can be omitted, saving space. In the above example, keys C and E have not issued any signatures, so they are omitted. Second, this makes the certificates smaller, so the certificate list is easier to read, easier to perform signatures on, and easier to pass around in the key management protocols.

3.2.2 Signatures

Every key in a key expression can perform a *validation signature* on its own certificate, or a *certification signature* on the next certificate in the chain. A validation signature expresses trust in the other keys in the key expression and implied agreement with any signatures they perform.

Since a single key can be used in multiple key expressions, a single cryptographic signature can perform multiple certification or validation signatures. In the above example, key D belongs to a validation authority, and it has issued a single cryptographic signature that functions as a validation signature within both certificates (since the signature's *listVA* attribute contains *"0"* for the first certificate and *"1"* for the second).

Every signature is calculated over the certificates list, starting with the first certificate and ending with the last certificate that is certified or validated by the signature. This ensures that every signature commits to all relevant context such as the cryptoID and the key variable bindings. In the above example, all three signatures are calculated over both certificates.

Now we see the necessity of storing the signatures separately from the certificates: First, a signature can refer to multiple certificates, so it wouldn't make sense to store a signature in one particular certificate. Second, since signatures cover all preceding certificates, if signatures were mixed into the certificates these signatures would be covered by later signatures, and these later signatures would break if the former were removed or re-issued.

We've glossed over many details of the certificate format, which can be found in section 3.3. of [2].

4. Key Management Protocols

Certification authorities (CAs) enable key replacement and cryptoID mobility. Validation authorities (VAs) enable key revocation. We now describe the protocols used with these key management servers.

4.1 Overview

Generally, a key management server can function either as a CA or a VA. A user will choose which servers to use, and in which capacity to use them.

The cryptoID protocols fall into either *control* or *operational* categories. The control protocols are used as necessary to configure and control key management servers. The operational protocols are used regularly to fetch the latest certificate chains and signatures.

In a *bottom-up* scenario, the user is responsible for choosing and controlling his own servers. In a *top-down* scenario, the user trusts some authority to choose and control the servers, which the user accesses only through the operational protocols.

The following steps are used to configure and control key management servers. In a bottom-up scenario, the client will be an end-user. In a top-down scenario, the client will be an administrator:

1. The client registers with different servers, establishing passwords that can be used for mutual authentication.
2. The client retrieves the keys belonging to the servers the client wishes to empower to issue certification or validation signatures.
3. The client incorporates these keys into a new certificate with a random certID value.
4. The client registers this certID value with the relevant servers.
5. The client signs the new certificate, and posts the entire certificate chain to the relevant CA servers.
6. The client revokes the certID with the relevant servers, if compromise or abuse of the certificate's keys is suspected.

The following operational steps are used by the end-user to acquire a valid certificate chain:

1. If the client already has a key pair and certificate chain stored locally, the client skips to step 5. Otherwise:
2. The client retrieves a previously-posted certificate chain from a CA server.
3. The client generates a key pair locally, and adds a certificate which references this key to the chain.
4. The client sends this new certificate to the CA server and receives back a signature on it.
5. The certificate chain may be annotated with a list of VA servers. If so, the client contacts these VA servers in order, sending each the list of certificates and receiving back a list of validation signatures. Once the client has received enough validation signatures to make the chain valid, the client is done.

4.2 Example Scenarios

We give a few examples of how key management servers could be used:

4.2.1 Alice

Alice is a salesperson for a software company. Alice's employer wants her to use secure communications with prospective clients, when possible. To enable this, Alice is given a cryptoID and an account on a CA server located on her employer's internal network.

Alice has instant messaging, voice, and email applications on her laptop. She configures these with the address of the CA. When she starts any of these applications, she is asked to enter her username and password. The application uses these to log in to the CA and acquire a certificate chain good for ten hours.

When Alice is on the road, she uses a VPN to get into the internal network and access the CA.

4.2.2 Bob

Bob is a political campaign manager. He stores his root private key in a smartcard locked in a safe. Once a month he uses this key to issue a certificate to a commercial CA service. His laptop, desktop, cellphone, home phone, and PDA are configured with the address of this service.

Bob instructs the service to issue certificates that expire within eight hours. Bob's devices will prompt him to re-authenticate and get a new certificate every four hours, so that even if the CA becomes unreachable for several hours Bob has a good chance of weathering the interruption without losing access to his cryptoID.

If a device is stolen, the thief will not be able to re-authenticate, and the device's certificate (if it has one) will become unusable within four to eight hours.

Bob worries about the commercial service being compromised. To handle that, the certificates he issues to the service require validation signatures by a second service. Bob instructs this service to issue validation signatures that expire within eight hours as well. Thus if Bob develops suspicions about the first service, he can revoke its certificate and shut it down within eight hours.

4.2.3 Carol

Carol participates in online chats on some controversial topic. She worries about her private keys being stolen and used to impersonate her.

To prevent this, she chooses three online servers as root CAs for her cryptoID, and requires certification signatures from any two of them. She instructs the CA servers to issue signatures good for ninety minutes. If one of her end-entity private keys is compromised, she can count on it expiring and being replaced within ninety minutes at most. If one of her CAs becomes temporarily unreachable, Carol can still get certification signatures from the other two.

4.2.4 Dave

Dave is in charge of deploying dozens of wireless access points across a university campus. Each access point will authenticate itself using the same cryptoID as the others (but a different certificate and subkey). A single cryptoID is used so students only have to configure their network settings with one cryptoID to authenticate any access point.

Dave is worried that an access point might be stolen or hacked. He decides that each access point will have to contact a university-run VA every 24 hours to get a new validation signature. If the administration learns of a compromised access point, its certificate will be revoked.

4.3 CertID Ownership

Key management servers keep track of "certID ownership" to prevent clients from tampering with each other. A client preparing a certificate will register its random certID with the relevant servers. Only *after* this will the client sign the certificate and use it in a way that makes its certID public.

The client who registered the certID is considered, by each server, to "own" the certID and any certificates containing it. Only the owner can revoke a certID. A CA server will not issue certificates under a revoked certificate, nor will a VA server issue validation signatures to a revoked certificate. CA servers will only issue certification signatures to clients who possess a "subordinate password" chosen by the owner of the issuing certificate.

The server will not allow anyone to re-register a certID until its previous registration has expired (even if it is revoked; in fact, *especially* if it is revoked). Clients should be polite and use registrations that expire, so that servers don't have to keep track of an ever-growing number of certIDs. Registrations should typically be set to expire when the certificate does.

4.4 Protocol Security

The protocols are secured with TLS. Some protocols require *mutual authentication*, whereas some only require *server authentication*.

SRP is used for mutual authentication based on a username and password [3]. CryptoIDs are used for server authentication – every server has a cryptoID and can use its certificate chain within TLS to authenticate itself.

Every user account has a *main password* used for registering and revoking certIDs, and posting certificate chains. CA accounts also have a *subordinate password*, used for retrieving posted certificate chains and requesting certification signatures. Using separate passwords allows the infrequently-used main password to be kept more securely. In a bottom-up scenario, the user chooses both passwords. In a top-down scenario, the administrator would typically choose both passwords and only reveal the subordinate one to the user.

Normally, the mutually-authenticated protocols just use SRP, and the server-authenticated protocols use cryptoIDs. However, the client can elect to perform mutual authentication with SRP *and* cryptoIDs simultaneously, or to perform server authentication with SRP (in which case the additional client authentication is superfluous but harmless).

4.5 VA Server Annotations

In Bob's scenario, the certificate chain retrieved from a CA needs a validation signature from a VA.

To indicate that a chain requires validation signatures, we add an `<annotations>` element as the last child of a `<certChain>`. The annotations are not strictly part of the chain, but contain arbitrary metadata about it. When a client posts a chain to a CA server, and later retrieves it, the annotations will be returned intact. The client must strip off the annotations before using the chain to authenticate itself.

The only annotation currently define is the `<vaServers>` element. This lists validation server URLs and cryptoIDs. To process this annotation, the client should go down the list and try to retrieve validation signatures from each listed server. Once the client has enough signatures to make the chain valid, the client stops. If a server cannot be contacted or fails for some reason, the client proceeds to the next server. If the client gets to the end of list and the certificate chain is still not valid, the application should notify the user and try again later.

The client should keep track of when validation signatures expire, and repeat the above process well before then (so if the VA servers become unreachable for some reason, the user has enough warning to do something about it).

```
<annotations>
    <vaServers>
        <vaServer cryptoID>+
    </vaServers>?
</annotations>
```

The *vaServers* element contains a list of *vaServer* children.

Each *vaServer* element gives a VA server's HTTPS URL.

The *cryptoID* attribute on a *vaServer* gives the cryptoID used to authenticate the server.

4.6 Protocol Interoperability

The different parts of the cryptoID system have different relevance to interoperability. The protocols are less important for interoperability than the data formats because a client can choose which key management servers to use.

The control protocols are less important for interoperability than the operational protocols because control of key management servers can be accomplished in other ways, such as through a web interface, or by offline communication with the administrator.

4.7 Protocol Summary

We list all of the protocols and their authentication requirements here for easy reference.

Common control protocols:

- *registerUser (server or mutual)*
- *getKey (server)*
- *registerCert (mutual)*
- *revokeCert (mutual)*

CA control protocol:

- *putCertChain (mutual)*

CA operational protocol:

- *getCertChain (mutual/subordinate)*

Common operational protocol:

- *getSignatures (mutual/subordinate or server)*

4.8 Common Features

Every protocol is a single request/response XML exchange, transported over HTTP POST and secured with TLS. We describe the XML protocol messages using the shorthand from section 3.3.2 of [2]. We reference some of the XML structures from that document as well.

Every response contains a `<result>` element as its first child.

```
<result>
    <code>
    <message>?
</result>
```

The *code* value may be *Success*, *RequesterError*, *ResponderError*, or some protocol-specific value.

The *message* value is only allowed if the code is not *Success*, and will give a more detailed explanation the client can use for debugging or logging purposes.

4.9 The RegisterUser Protocol

This protocol registers a username and SRP verifier with a CA or VA server. The protocol requires server authentication for a first-time registration. Subsequent registrations, used to change the account password, require mutual authentication with the main password.

When registering with a CA server, the client may register a subordinate password along with the main password. The subordinate password may be used instead of the main one for the *getCertChain* and *getSignatures* protocols.

```
<registerUserRequest>
    <username>
    <verifierEntry>
    <verifierEntry>?
</registerUserRequest>

<verifierEntry>
    <N>
    <g>
    <salt>
    <verifier>
</verifierEntry>

<registerUserResponse>
    <result>
</registerUserResponse>
```

The *username* value is a username string.

The *verifierEntry* elements contain *N*, *g*, *salt*, and *verifier* values which are base64-encoded SRP parameters; see [3] for details. When two *verifierEntry* elements are present, the first is for the main password, and the second is for the subordinate. The subordinate verifier is calculated using the string "subordinate:" prepended to the username.

The *UserRegistered* result code will be returned if the username is already registered.

4.10 The GetKey Protocol

This protocol retrieves a CA or VA server's signing key. Only server authentication is required.

```
<getKeyRequest>
</getKeyRequest>

<getKeyResponse>
    <result>
    <publicKey>?
</getKeyResponse>
```

A *publicKey* will be returned if successful.

4.11 The RegisterCert Protocol

This protocol registers a certID with a key management server. The client must authenticate with SRP.

```
<registerCertRequest>
    <type>
    <certID>
    <notAfterDelta>
    <expiresOn>?
</registerCertRequest>

<registerCertResponse>
    <result>
</registerCertResponse>
```

The *type* value must be "CA" or "VA", and specifies whether this certID is being registered for the purpose of certification or validation signatures.

The *certID* value identifies the certificate being registered.

The *notAfterDelta* value specifies the number of minutes the server will add to the current time when determining the expiration time for certification or validation signatures.

The *expiresOn* value, if present, specifies when this registration should be deleted. This value is typically set to the certificate's *notAfter* time, if it has one. This value allows the server to get rid of database entries that are no longer needed.

The *CertRegistered* result code will be returned if this certID has already been registered, whether by this or any other user.

4.12 The RevokeCert Protocol

This protocol revokes a certID with a key management server. The client must authenticate with SRP. A certID can only be revoked by the same user who registered it.

```
<revokeCertRequest>
    <certID>
</revokeCertRequest>

<revokeCertResponse>
    <result>
</revokeCertResponse>
```

The *certID* value identifies the certificate being revoked.

The *CertNotRegistered* result code will be returned if this certID is not registered with this server.

The *CertRevoked* result code will be returned if this certID has already been revoked.

The *NotAuthorized* result code will be returned if the client is not authorized to revoke this certID.

4.13 The PutCertChain Protocol

This protocol posts a certificate chain to a CA server. The client must authenticate with SRP.

The certificate chain will be posted to the client's account. An account can only have one chain posted at a time; if the client posts another chain, it will overwrite the previous one.

```
<putCertChainRequest>
    <certChain>
</putCertChainRequest>

<putCertChainResponse>
    <result>
</putCertChainResponse>
```

The *certChain* value is a cryptoID certificate chain.

4.14 The GetCertChain Protocol

This protocol retrieves a posted certificate chain from a CA server. The client must authenticate with SRP. The subordinate password may be used instead of the main one. In this case, the string "subordinate:" must be prepended to the username when authenticating, so the server knows to authenticate against the subordinate password.

A certificate chain must have been previously posted to the client's account.

```
<getCertChainRequest>
</getCertChainRequest>

<getCertChainResponse>
    <result>
    <certChain>?
</getCertChainResponse>
```

A *certChain* will be returned if successful.

The *NoCertChain* result code will be returned if a certificate chain has not been posted to the client's account.

4.15 The GetSignatures Protocol

This protocol retrieves signatures from a key management server. Only server authentication is required to retrieve validation signatures. Client authentication (with the main or subordinate password) is required to retrieve certification signatures.

```
<getSignaturesRequest>
    <certs>
</getSignaturesRequest>

<getSignaturesResponse>
    <result>
    <signatures>?
</getSignaturesResponse>
```

The *certs* value is an entire sequence of certificates taken from a *certChain*. The server will issue signatures on certificates in this sequence which have a certID that is registered and not revoked, and which reference the server's key.

The server will issue certification or validation signatures (or a combination) depending on how the different certIDs were registered. The server will refuse to issue certification signatures unless the client has authenticated with the main password or subordinate password associated with the account that registered the certID.

A *signatures* element will be returned if successful. This is a list of signatures that the client should add to his certificate chain.

The *NoSignatures* result code will be returned if the server doesn't find any certificates to sign.

5. AVAILABILITY

The cryptoIDlib python library implements the cryptoID system:

http://trevp.net/cryptoID/

6. REFERENCES

[1] T. Aura. Cryptographically Generated Addresses (CGA). *To Appear in Information Security Conference 2003*, 2003. *http://research.microsoft.com/users/tuomaura/CGA/*

[2] T. Perrin. Public Key Distribution through "cryptoIDs". *New Security Paradigms Workshop*, 2003. *http://trevp.net/cryptoID/cryptoID.pdf*

[3] D. Taylor, T. Wu, N. Mavroyanopoulos, and T. Perrin. Internet-Draft: Using SRP for TLS Authentication, January 2004. *http://www.ietf.org/internet-drafts/draft-ietf-tls-srp-06.txt*