Public Key Distribution through "cryptoIDs"*

Trevor Perrin
trevp@trevp.net

ABSTRACT

In this paper, we argue that person-to-person key distribution is best accomplished with a key-centric approach, instead of PKI: users should distribute public key fingerprints in the same way they distribute phone numbers, postal addresses, and the like. To make this work, fingerprints need to be *small*, so users can handle them easily; *multipurpose*, so only a single fingerprint is needed for each user; and *long-lived*, so fingerprints don't have to be frequently redistributed. We show how these qualities can be achieved with simple and well-understood techniques. The chief technique is for each user to store a *root key* in a highly secure environment and use it to certify *subkeys* for use in more convenient environments. Certificate formats like X.509, PGP, and SPKI could be used for this, but we argue that a format designed expressly for this could do a better job; thus we design the *cryptoID certificate format*.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection – *authentication*; E.3 [**Data Encryption**]

General Terms

Design, Human Factors, Standardization.

Keywords

Public key infrastructure, key distribution, key management, fingerprints, cryptoIDs.

1. INTRODUCTION

When Diffie and Hellman invented public key cryptography, they assumed public keys could be distributed by placing them "in a public directory along with the user's name and address" [10]. Implicit in this is the idea that public keys are similar to addresses, and thus can be distributed in the same fashion (such as through directories like phone books, or through manual methods like exchanging business cards).

This approach to key distribution is particularly apt for securing person-to-person communications, since users are already accustomed to exchanging and managing addresses in this setting. Thus, the metaphor "public keys as addresses" allows us to repurpose existing infrastructure (directories, address books, etc.) and existing user behaviors for key distribution.

*** This is a revision of the paper originally presented at NSPW 2003.**

There are a few problems with treating public keys as addresses:

- Public keys are *large*: thus they are difficult for people to read, write, speak, memorize, or compare.
- Public keys are *single-use*: it is good practice to use different key pairs for different protocols or in different devices, and thus a person will have multiple public keys.
- Public keys are *transient*: they need to be revoked when the private key is lost or stolen, and should be changed periodically.

It would be burdensome to transmit multiple large public keys, frequently, from the subject to the relying party. Thus we must construct some sort of *cryptographic identifier* which is like a public key but is *small*, *multipurpose*, and *long-lived*:

- *Small:* Through *hash extension* [1] and base 32 encoding we can construct a public key fingerprint that is half the length of current fingerprints (20 versus 40 characters) but still achieves an adequate security level.
- *Multipurpose:* Each user's fingerprint would correspond to a *root key* under which the user certifies the *subkeys* he uses in particular protocols, or on particular devices.
- *Long-Lived:* The root key would be kept in a highly secure environment, possibly in the possession of some trusted party or threshold of parties. Root key compromise would thus be a rare occurrence, and subkey compromise would be dealt with by frequently re-issuing and revalidating subkey certificates.

The certificates issued by a root key to its subkeys are neither identity certificates (like X.509 [25,28] or PGP [6,27]) nor authorization certificates (like SPKI [13,15]). We call them *key management certificates*. These certificates wouldn't contain identity or authorization data, but they would benefit from techniques such as *threshold subjects* [16] and *timed revalidation* [17]. X.509, PGP, or SPKI certificates could be used for key management, but a certificate system designed specifically for this could be much simpler, and could implement the above techniques more effectively.

In what follows, we consider key infrastructure as having two parts: public key distribution and private key management. In section 2 we examine public key distribution, and argue against identity certificates in favor of *address-based key distribution*. In section 3 we consider private key management and argue for *key management certificates.* Finally, since current certificate formats are not ideal for key management, we design the *cryptoID certificate format*. The fingerprints used with this format we call *cryptographic identifiers*, or *cryptoIDs*.

2. PUBLIC KEY DISTRIBUTION

2.1 Scope of the Problem

Public key distribution is the problem of distributing to parties, in a trustworthy fashion, the public keys of those with whom they wish to communicate. This problem must be pursued in a wide range of settings, each with its own requirements, constraints, and opportunities.

The setting we are interested in is end-to-end security for person-to-person communications. We would like a solution that could be built into consumer hardware and software and which would enable widespread use of authenticated and confidential phone calls, emails, text messaging, video conferencing, and the like, whether in business, personal, national security, or any other environment.

Such a solution would meet three primary requirements:

- *Flexibility as to degree of assurance:* Users in different environments will have different security requirements. When chatting with her Mom, Alice might not care much if the public key is authentic. When arranging a high-value money transfer, she will care greatly. A one-size-fits-all solution will be too burdensome in some cases yet too lax in others. Achieving a low assurance level must be easy, and achieving a high one must be possible.

- *Flexibility as to means of assurance:* Users in different environments will find different means of gaining assurance more natural. When users have frequent real-world contact and are not part of any overarching security domain, manual exchange of fingerprints is the most feasible approach. When users are part of the same security domain, a trusted authority for this domain should be able to introduce them to each other. When users belong to different domains, it should be possible to link these domains through some sort of delegation, and in this fashion assemble a large-scale infrastructure.

- *Simplicity*: If we expect security analysts to review and approve of the solution, programmers to implement it, administrators to deploy and manage it, and users to comprehend and use it, it must be incredibly simple. In a sense this is obvious. However, engineering always runs into trade-offs, and we are declaring up-front our bias for simplicity over efficiency, feature-richness, backwards-compatibility, extensibility to different use cases, or other desiderata.

X.509-based Public Key Infrastructure (henceforth just "PKI") and PGP are commonly offered as solutions to this problem. In 2.2 we assess these systems against our requirements and find them lacking. In 2.3 we present an alternative approach.

2.2 Current Approaches

PKI and PGP both use certificates for key distribution – PKI relies entirely on certificates, whereas PGP also allows direct key exchange, verified by fingerprints. Nonetheless, to fully understand both systems, we must understand certificates.

A certificate is a document, signed by the issuer's public key, which contains the subject's public key and some identity or authorization information. It can be viewed as an assertion *by* the issuing key *about* the subject key. Since certificates are small and self-contained, they can be passed around and re-used easily, and since they are signed, they can be distributed over untrusted channels. A *offline trusted third-party* Trent can thus express his opinion *once* that Alice's public key really belongs to her, and Alice can publish this opinion in untrusted directories, and pass it around by herself, to convince anyone who trusts Trent of the association between herself and her public key.

This sort of certificate, which binds a public key to a name or address, is an *identity certificate*. An *authorization certificate* binds a public key to an authorization, such as the right to access a particular server, or read a particular file. Authorization systems such as SPKI are used primarily for access control; for the person-to-person scenario, we will be concerned with identity systems such as PKI or PGP.

We can further divide identity certificate systems into *simple* and *networked* systems. A simple system, like our scenario with Alice, Bob, and Trent, has only a single intermediary between the subject and the relying party. In a networked system there are *end-entity* certificates, which we've been discussing, and *certificate authority*, or *CA*, certificates. These latter grant the subject the power to issue certificates himself for some set of names. The subject could delegate these powers further by issuing his own CA certificates, and so on. The result is that Bob will choose some certificate authorities as his *trust anchors*, these will certify some other CAs, which will certify some other CAs, and eventually one will certify Alice. To acquire trust in Alice's public key, Alice presents, or Bob discovers, a certificate path from one of Bob's trust anchors to Alice's public key.

PKI is a networked system. The PGP web of trust can be used as a networked system but is perhaps better viewed as a *network of simple systems*, since people sign each other's keys, but they usually don't delegate a specific portion of the namespace to the signee, nor does software automatically discover paths. As we mentioned, PGP also allows fingerprint-verified key exchanges - thus PGP key distribution is a hybrid of fingerprint and certificate techniques.

We now show how PKI and PGP fall short of our requirements.

2.2.1 PKI vs. Requirements

Most PKI applications try to automate key distribution completely – when a valid certificate path is discovered for a communication, some icon is shown to indicate security. Since software can't know the user's security requirements for this particular communication, this is necessarily a one-size-fits-all approach. At best, the user can inspect the certificate path and decide whether the issuers, policy OIDs, and Certificate Practice Statements [25] are satisfactory – yet most users have no idea what these things are. Thus, PKI offers little flexibility for the average user to achieve different degrees of assurance in different circumstances.

This automation also means PKI has little flexibility with respect to *means* of assurance. If Alice and Bob have frequent real-world contact, it may be quite easy for Alice to give Bob her fingerprint manually. Yet PKI tries to get by without involving the user, so

it has no way to exploit this relationship. Instead, Alice and Bob *must* exchange keys through the PKI infrastructure, since this is all the software knows about. This requires Alice to authenticate with Trent and procure a certificate, and it requires Bob to add Trent as a trust anchor, and to ensure that both he and Trent share the same name for Alice.

If we draw the relationship between Alice, Bob, and Trent as a triangle, scaled to reflect ease of key distribution, Alice and Bob may be close and Trent may be quite distant, so *requiring* all key distribution to occur through third parties is inefficient. Alice and Bob can work around this if Alice becomes her own CA and issues herself a certificate, but then Bob has to treat Alice's CA as a trust anchor, which is a huge security risk.

If PKI isn't flexible enough to support manual fingerprint exchange, we still might expect it to be flexible in supporting delegation amongst trusted third parties. A CA can delegate authority to another just by issuing a CA certificate for some set of names. Since these certificates can form any sort of topology, this *seems* a very flexible system. However, this form of delegation is extremely limited, since the issuing CA can only delegate authority over concisely representable portions of the namespace. When the desired delegations can't be exactly expressed as permitted and excluded subtrees [25] of some namespace (such as X.500 or DNS), the issuing CA is faced with the procrustean choice of:

- Delegating too little authority and disenfranchising some users.
- Delegating too much authority and creating a security risk.
- Issuing multiple certificates, thus complicating path construction.

This is a specific instance of a general problem – with certificates, all delegations must be *expressed* in some language which is necessarily limited. We can see the problem clearly by considering a more flexible form of delegation - *trusted directories*. A trusted directory is an *online trusted third party* which clients query for public keys. This is what Diffie and Hellman started with, and what many, even in the PKI community, are returning to (under the name *validation servers* - see SCVP [33,39] or XKMS [24]).

A trusted directory supports more flexible forms of delegation than a CA because a trusted directory *implements* its delegations instead of just *proclaiming* them. Thus, while a CA is limited to delegations which can be expressed in a certificate and processed by client software, a trusted directory can do *anything* when a client asks it for a key – it can look in its local storage, consult another directory, consult a PKI, consult *multiple* other directories and PKIs and corroborate the results, and so on.

Trusted directories are not only more *flexible* than CAs, but they are *simple* for clients to use – a client just consults a directory and receives an answer. CAs, by constrast, simply proclaim that they are delegating some authority and then wash their hands of it. It is left to the client to find all the relevant proclamations and make sense of them. Clients must pull certificates from a wide range of sources while trying to discover a path through the certificate graph for which the delegations are valid, the signatures are correct, and the certificates have not been revoked. This is an extraordinarily complex problem [8,23], so placing this responsibility on client software, which is widely deployed and thus difficult to upgrade and configure appropriately, is a poor design choice.

CAs are more complicated than trusted directories in another way. Consider the triangle between Alice, Bob, and Trent. If Trent wants to tell Bob what Alice's public key is, the simplest information flow would be directly from Trent to Bob. But if Trent is a CA, he is offline and thus can't speak directly to Bob. Instead, Trent must communicate with Bob in a roundabout fashion – he must enlist *Alice* to present a certificate to Bob. Thus, the offline nature of CAs forces end entities to mediate communications between CAs and relying parties.

If *online trusted third parties* (i.e. trusted directories) are so much more flexible and simpler than *offline trusted third parties* (i.e. CAs), why are CAs the conventional approach? Offline TTPs are seen as having three advantages:

- *Performance*: The TTP is not a communication bottleneck.
- *Partition Resilience*: The system will keep functioning if the TTP becomes unreachable.
- *Private Key Protection*: The TTP's private key can be kept offline, in a more secure environment.

These are legitimate performance and robustness goals, but they aren't worth the crippling complexity CAs impose. Furthermore, CAs are not even succesful in achieving the first two benefits: relying parties desire timely assurance of a key's authenticity, and this requires an *online* mechanism be added back in – either the subject or the relying party needs to frequently contact some online party to procure validation data.

In sum, PKI makes two mistakes with respect to key distribution. First, it ignores the user and tries to automate everything: it neither takes account of the variability in users' security requirements nor takes advantage of the user's capacity for manual key distribution. Second, PKI pays an enormous price in limitations and complexity for the marginal benefit of offline TTPs.

2.2.2 PGP vs. Requirements

PGP allows users to exchange keys over untrusted channels such as key servers or web pages, and then verify the keys against trusted fingerprints. We have two criticisms of this use of fingerprints:

- *Key-Centric instead of Fingerprint-Centric*: PGP uses fingerprints only to verify keys, instead of treating fingerprints as the primary element of key distribution. A PGP end-user cannot simply enter a fingerprint and address into his software with the meaning "use this fingerprint for this address". Instead he has to *acquire* the key, *import* the key onto his key ring [47], *verify* the key's fingerprint, and then *sign* the key to record that it is trusted.
- *Size*: PGP fingerprints are 32 or 40 characters long when hex-encoded. This makes them difficult to

display in business cards and paper directories, and difficult for people to deal with.

We also have a nit:

- *Presentation*: PGP Fingerprints aren't written in a consistent form – different users call them different things, and often include details which shouldn't be necessary such as key ID, key type, or key size. This makes fingerprint handling confusing, particularly to novice users.

PGP also allows indirect key distribution through a *web of trust*. In this model, people certify each other's keys, and each relying party assigns trust levels to keys belonging to certain people. When presented with a new key, relying party software will check whether the new key has been certified by keys whose trust levels sum to a sufficient threshold; if so, the new key will be considered valid.

Certificates are a much better design choice for PGP than PKI: the PGP web of trust consists of a network of *people* functioning as trusted third parties. Since people cannot be expected to be *online*, serving at the beck and call of relying parties, certificates are a necessity. Nonetheless, certificates introduce complexity and rigidity into PGP just as in PKI.

Complexity because the relying party is forced to explain his trust decisions to the computer. To do this, he must understand not only the cryptographic concepts of keys, signatures, and certificates, but also the web of trust concepts of trust and validity, and how the latter is calculated from the former [45,46]. Furthermore, he must understand how to manipulate these concepts by importing and signing keys, assigning trust levels, and reading validity levels. We emphasize that the underlying concepts of trust and validity are not complex – people deal *intuitively* with them every day. The problem is that PGP forces users to deal *quantitatively* with these fuzzy notions and to manage them across the human-computer interface.

Rigidity is introduced because the user must express his trust decisions in a limited language. Real-world trust is above all *context-dependent*. Bob and Charlie may be partners in business but competitors for the fair hand of Alice. Alice may be an employee of Bob but a union organizer on the side. Alice may trust Charlie in little things, yet know he has an ethical weak spot when it comes to making a buck. Questions of trust can't be answered in the abstract, yet PGP assumes a user can assign fixed trust levels to individuals, and that these trust levels will interact additively. These are hugely simplistic assumptions.

Besides trust management, another source of limitation and complexity in the web of trust is path discovery. In PKI this complexity is felt by end-entity software, whereas in PGP it is felt by the user. This is because the web of trust has no structure that could guide automated path discovery, so software doesn't even try. If a user is presented with a new key which none of his current trusted keys have certified, it is up to the user to download keys from the keyserver until he finds a trusted path to the new key. This is neither scaleable nor convenient.

Finally, we point out that PGP is usually delivered as a stand-alone application. This is of little use to end-users, who need security within the devices and applications they actually use. Trying to splice PGP support into these environments through plugins typically results in a poor user experience. Why haven't software vendors added PGP support to their applications? We surmise that the complexity of PGP trust management has made this task seem too imposing. A public key distribution system that aims for wide deployment must make life simple for *developers* as well as end-users.

In sum: Like PKI, the PGP web of trust tries to automate calculations best performed in the user's mind. The result is excessive complexity for both users and developers [44]. Furthermore, a web of trust provides no basis for path discovery, so it's unclear how this could scale beyond small, tight-knit communities. On the positive side, PGP users can ignore the web of trust and just use fingerprints to verify a key exchange, but this is still excessively complex.

In the next section, we show how fingerprint exchange by itself can meet our requirements.

2.3 Address-Based Key Distribution

We argue that fingerprint exchange could form a person-to-person key distribution architecture that is both *flexible as to degree and means of assurance* and *simple*. This argument is based on an idealized notion of fingerprints which are:

- *Small*: We assume fingerprints that are around 20 characters in length.
- *Multipurpose*: We assume that each fingerprint corresponds to multiple subkeys which can be used in different protocols on different devices.
- *Long-Lived*: We assume that each fingerprint corresponds to a root key that is managed in a highly secure fashion, so revocation due to key compromise is rare.

In section 3 we show how to approximate these idealized fingerprints through real-world techniques. For now, we show how such fingerprints would facilitate key distribution.

2.3.1 Fingerprints as Addresses

We propose that users can be educated to understand that acquiring the fingerprint of another party is necessary for secure communication with that party. Furthermore, it should be explained that the fingerprint must be acquired through a secure channel, lest they be tricked into using a fingerprint belonging to someone else.

Armed only with this knowledge, and with their intuitive grasp of all the ways in which small tokens of data can be communicated, we believe that users could take key distribution into their own hands. This *human-centric* approach to key distribution would yield a system much more flexible, and much simpler, than trying to automate key distribution through PKI or the PGP web of trust.

The key to this argument is the metaphor *fingerprints as addresses*. Like phone numbers, postal addresses, email addresses, and the like, fingerprints are small pieces of data that must be exchanged as a prelude to communication.

It can be objected that fingerprints aren't small enough to justify the metaphor – an X.509 or PGP v4 fingerprint is a hex-encoded SHA-1 output, which works out to 40 characters. In response, we will later design 20 character fingerprints that look like these:

```
f3v4g.ifcen.r3rj5.cmbx8
```

```
eg9zk.yv89c.yk4kr.dufge
```

```
bf45a.qssfo.5ur8z.cx3ba
```

These are a little longer than most email addresses, a little shorter than most postal addresses, and about the same size as a credit card number plus its 4-digit expiration date. Since users are capable of managing these items, we contend that users are capable of managing fingerprints.

Fingerprint distribution would allow users to acquire assurance in a fingerprint through any channel available to them: fingerprints can be printed on business cards; written on napkins; read aloud over the phone, over the radio, or in person; sent in email or postal mail; published in paper or electronic directories, print advertisements, or web pages; exchanged on removable media; or handled through any channel that users find convenient. Thus we achieve *flexibility as to means of assurance.*

Since users would be directly involved in acquiring and corroborating fingerprints from different sources, users can mentally estimate the degree of assurance they have in any fingerprint. If this degree is insufficient for the desired use, they can consult other sources, give up the attempt to communicate, or proceed with the communication while retaining some suspicion. Thus we achieve *flexibility as to degree of assurance.*

Since users can view fingerprints as a sort of "crypto address", they should be quite comfortable exchanging them, importing them into address books, notifying people when they change, and so on. In contrast, schemes which try to simplify things through automation end up forcing users to grapple with the concepts of keys, signatures, certificates, certificate chains, revocation lists, trust roots, and validity/trust calculations. Paradoxically, by giving users *more* responsibility we make things *easier* for them, since less automation means less machinery and fewer concepts to deal with. Thus we achieve *simplicity for end-users*.

With PKI, end-user software must perform path discovery and validation. With PGP, software must provide a key management interface. Both approaches make it difficult for developers to add communication security to their products. However, almost all personal communications software has an address book. It would be easy to add a new field to address book entries to contain the specified person's fingerprint. When a communication is authenticated to a fingerprint that matches some entry, the only thing software would have to do is display the name of the entry (i.e. the *pet name* [35] "Mom", "Bob", etc.), and an authentication indicator. Thus we achieve *simplicity for application developers.*

Fingerprint distribution could piggyback on address distribution: business cards and trusted directories would carry addresses and fingerprints in tandem, and if you could get someone's address from a mutual friend, you could probably get their fingerprint as well (the *Granovetter diagram* models this type of interaction; see [7,36]). In corporate environments, the enterprise directory could deliver fingerprints along with addresses. Since pre-existing address distribution infrastructure can be re-used for key distribution, we achieve *simplicity of infrastructure.*

We now remark on some complications that result from the *fingerprints-as-addresses* approach:

2.3.2 Retrieving Encryption Keys

Alice cannot encrypt an email to Bob if all she has is his fingerprint. PGP requires Alice to either:

- Find Bob's key and import it manually, *or* –
- Download the key from an appropriate keyserver.

The first approach requires too much effort on Alice's part, and the second depends upon a global, universally agreed-upon infrastructure. Either approach requires Alice to carry around her key ring if she wishes to encrypt to Bob from different computers.

We would prefer to have Bob's trust information in a more concise form that could be stored in directories and address books, so that Alice could import this information from a directory into her address book, and synchronize her address books, without having to juggle keys and certificates. We'd also like to streamline the conveyance of this information, so that Alice doesn't have to acquire it manually, or depend on any global infrastructure.

One approach is to have each user choose his own *key URL*, where he will post the latest version of his encryption keys. Address books and directories could have an additional field for this URL. All of Bob's secure communications would contain his key URL, so that Alice's software could populate this field automatically. Where requesting a *bootstrap message* from Bob is inconvenient or impossible, Alice can fill in this field herself.

2.3.3 Proof of Possession

Alice could lie and claim ownership of Bob's fingerprint. To someone who believed her, Bob's communications would appear as if they were originating from Alice. If you only retrieve fingerprints from people or directories you have a great deal of trust in, then this is not a concern. Otherwise, the only way to prevent this is to check and make sure that Alice can be contacted under the fingerprint she claims is hers.

PGP provides this check, to some extent, since every key should have a self-signed User ID containing the name and email address of the key's owner. If you can recognize that these match Alice and no-one else, then you can feel assured that Alice is not trying to claim someone else's key.

We don't wish to follow this approach – it means that a relying party must retrieve and inspect a user's key before asserting trust in the fingerprint, and that would complicate fingerprint distribution. Instead, we suggest warning users that people might lie about their fingerprint, so you should contact them under their claimed fingerprint to make sure it's really theirs (software could provide this warning automatically).

In practice, this check will happen in the course of things, since most communications contain identifying information, either *explicitly* (in message headers that identity the sender) or *implicitly* (in the contents of the communication itself).

2.3.4 Human-Readable Certificates

We have pointed out that certificates make sense in PGP, since they allow Trent to express his opinion about Alice's public key in a form that Alice can re-use. If we're giving up certificates, it seems that any relying party who trusts Trent will have to contact him directly and ask him about Alice's fingerprint.

However, Trent could easily send Alice a signed email, where he writes "I, Trent, hereby certify that the fingerprint `dbmv6.6zpre.wahq4.gqzjz` belongs to Alice." Essentially, this would be a human-readable instead of a machine-readable certificate (we could call it a *letter of introduction*). Trent could describe himself and Alice in any way that might be meaningful to a relying party (mentioning nicknames, physical descriptions, places of residence, etc.). Similarly, he could describe how he authenticated Alice. Alice could forward this email to relying parties to convince them of her fingerprint's authenticity.

2.3.5 Revocation

Suppose Bob receives Alice's fingerprint from Trent. Later, Trent discovers that he gave Bob the wrong fingerprint. Trent would like to *revoke* the fingerprint that he's already distributed to Bob.

If Trent is a trusted directory, then Bob can poll Trent on a regular basis. If Trent is a person, this can't be done as easily. Nonetheless, if Bob and Trent have at least occasional contact, then Bob would expect to learn from Trent if Alice's fingerprint has changed. If the revocation is important enough, Trent may broadcast it through channels that will reach most relying parties (such as mailing lists, web sites, phone trees, etc.).

Thus, instead of designing a technical infrastructure for revocation notices, we will assume these can be distributed through the same mix of infrastructures that distributed addresses and fingerprints in the first place.

2.3.6 Untrusted Fingerprints

Communication devices should always encrypt and authenticate themselves, even when the fingerprint of the other party is unrecognized. *Opportunistic encryption* of this sort protects against passive eavesdroppers, even though it doesn't authenticate the other party.

Software could automatically populate address book entries with unrecognized fingerprints lifted from communications. These fingerprints should be prefixed with some marker such as "(untrusted)". When a communication is authenticated to an untrusted fingerprint, software should *not* display an authentication indicator. However, if a communication from the same address fails to match the fingerprint, software *should* display a warning about the mismatch.

A user could verify an untrusted fingerprint and then remove the "(untrusted)" marker. Since it's easier to compare fingerprints than to type them in, this would save the user some effort.

2.3.7 Conclusion

We have tried to solve key distribution by reducing it to address distribution – fingerprints are *like* addresses, and address distribution is a solved problem. However, we've forced the analogy by *assuming* that fingerprints could be made small, multipurpose, and long-lived. Now we must show how to do so.

3. PRIVATE KEY MANAGEMENT

3.1 Scope of the Problem

Our approach to key distribution was based upon fingerprints. However, a fingerprint that is simply a hash of a single public key is quite limited:

- Alice may have multiple communications devices (a desktop, a laptop, a cellphone, and a PDA, for example), and each device may speak multiple protocols. Sharing a single private key amongst all these devices and protocols would be cumbersome and risky (both in the act of sharing, and in the fact that a compromise of one device or protocol would compromise them all).
- Alice faces a tension between keeping her private key *secure* and keeping it *accessible*: From the perspective of security, Alice would love to bury her private key in a mineshaft. However, Alice also wants to *use* her private key on all her devices and in all different circumstances – at home or at work, when travelling, and so on.
- If the private key is lost or stolen, Alice has no way to recover short of creating a new key pair and redistributing the fingerprint.

In discussing key distribution we assumed that each person has a single, long-lived fingerprint. Now we must consider how to create such a fingerprint which also supports:

- Multiple private keys per fingerprint.
- Robust security and accessibility of private keys.
- Recovery from loss or theft of private keys.

We call this the *private key management* problem. Our solution is for Alice to distribute the fingerprint of her *root key*. Alice will then use her root key to issue certificates to the *subkeys* she uses for particular protocols and devices. These certificates will limit the allowed uses of subkeys and will specify revalidation requirements so that subkeys can be disabled if compromised. Whenever Alice communicates with Bob, she will present the certificate chain from her root key to the subkey she is using, and Bob will validate the chain against Alice's fingerprint.

Thus we accomplish:

- *Multiple private keys per fingerprint*: Alice can use a different subkey for each protocol on each device.
- *Robust security and accessibility of private keys*: Since the root private key only needs to be used ocassionally, it can be kept in a highly secure manner. The less important subkeys can be made accessible to different devices.
- *Recovery from loss or theft of private keys*: Subkey certificates will have short lifetimes and revalidation requirements, so that theft of the private key can be quickly recovered from.

The certificates used for private key management are different from the identity and authorization certificates used for public key distribution, and so we will call them *key management*

certificates. Whereas identity or authorization certificates bind the key to some entity and must describe who this entity is, or what he is allowed to do, key management certificates assume that the root key is already bound to an entity. Thus, they only address the simpler problems of binding a subkey to the root key, and confining the damage done if the subkey is compromised.

This approach to key management is not novel. The PKI, PGP, and SPKI communities have all considered how their respective certificate formats could be adapted for private key management [43,3,19]. In the next few subsections, we consider the use cases and requirements for key management certificates. In 3.2 we examine PKI, PGP, and SPKI key management certificates, and conclude that none of them meet our requirements. In 3.3 we present the cryptoID certificate format.

3.1.1 Rootholders and Certificate Servers

Root keys should be stored in tamper-resistant hardware and kept under lock and key. Most users would be unwilling or unable to manage their root key this securely. These users should choose some trusted party as their *rootholder*.

A rootholder is a self-chosen, offline CA. The rootholder would keep the root key physically secure, and would issue subkey certificates to an online CA, or *certificate server*. The user would periodically authenticate to the certificate server, probably using a mutually-authenticated password protocol such as TLS/SRP [42], and the server would issue short-lived subkey certificates to the user. Since the certificate server would be online, the user could retrieve certificates from anywhere, whether at home, at work, or travelling, using any of his devices.

Business users would access the rootholder and certificate server infrastructure maintained by their employer. Home users would be free to select their own rootholders and certificate servers. They might choose a fee-based commercial service, a free service that comes bundled with something else (such as their operating system, cell phone, or internet connection), or a nonprofit service offering key management infrastructure as a benefit to the community.

A user might not have sufficient trust in any of these parties. To eliminate any single rootholder or server as the sole point of failure, users should be allowed to choose a root key or subkey as a *threshold subject* [16] of different keys held by different parties. A user can thus assemble a *trusted aggregate* out of several partially-trusted services which he feels are unlikely to collude, or to suffer a common-mode failure.

3.1.2 Certificate Lifetimes and Timed Revalidation

The lifetimes of subkey certificates should be chosen so as to balance two security risks. Frequent re-issuance of certificates allows frequent replacement of the *subject's private key*, making each key a less valuable target for cryptanalysis or theft. However, frequent re-issuance increases the exposure of the *issuer's private key* to theft or misuse.

Because of this tension, it is unlikely that we can re-issue certificates as frequently as we would like. However, we can achieve some of the benefits of frequent re-issuance without the costs by requiring *timed revalidation* [17] of certificates. The issuer of a subkey certificate can nominate a *validation authority* (or VA) by including the VA's public key in the certificate. The certificate will only be considered valid when presented in conjunction with a signature from the VA, and the VA can include expiration times in these signatures (if the VA includes a nonce instead of an expiration time, then we have *one-time revalidations* [18]; these have some advantages, but not enough to justify the added complexity). If the VA receives notice that the subject's private key has been stolen, the VA will refuse to issue further validation signatures and the certificate will become unusable once its current signature expires.

Revalidations don't allow the subject to change private keys, but they also don't require the involvement and exposure of the issuer's private key; instead, the VA's key is involved. This key is less important, since the VA can't issue certificates, and since a compromised VA can be removed the next time a certificate is issued.

Frequent revalidations introduce another security risk, and another trade-off. If a user is prevented from contacting the VA, the validation signature will expire and the subkey will become unusable. There is a trade-off here between *safety* and *liveness*: the more frequently the user has to revalidate, the more quickly a compromised key can be shut down, but the more likely it becomes that a *denial-of-service* attack or network failure can prevent use of a legitimate key.

To manage this trade-off, a certificate should be able to specify the VA as a threshold subject. To increase liveness, different VAs can be linked through *disjunctions*, so that if one is unreachable the user can contact another. To increase safety, different VAs can be linked through *conjunctions*, so that if one is compromised, the other can disable the subkey on its own. Through these techniques a validation infrastructure can be tailored to different requirements.

Example: Alice doesn't trust anyone else to hold her root key, but she is worried it might be compromised. To mitigate this risk, Alice can choose an online service as the VA for her root key. If the online service misbehaves it can disable Alice's root key, but it cannot impersonate her [20]. To mitigate the risk of service misbehavior, Alice can choose a backup VA, so that both VAs have to fail to prevent Alice from using her root key.

3.1.3 Time Synchronization

The use of short-lived certificates and revalidations requires relying parties to keep accurate time. Personal computers and devices often have no reliable, trustworthy source of time. Nonetheless, the only way to avoid time synchronization would be for every communication to be supplemented by online exchanges with the issuing and validating parties. The performance and denial-of-service implications of this are severe. In contrast, keeping accurate time should only require *occassional* contact with a trusted time source.

If we assume that relying parties can stay within at least 5 minutes of the correct time, then certificates and validation signatures that expire in less than 5 minutes should not be used.

3.1.4 Requirements

We can review the above scenarios and extract some requirements for key management certificates:

- An authorization language for saying which protocol(s) a key can be used with (voice, instant messaging, etc.).
- Threshold subjects
- Timed revalidation
- Threshold VAs
- Simplicity and security

In the next section, we argue that current certificate formats such as PKI, PGP, and SPKI do not meet our requirements.

3.2 Current Approaches

PKI, PGP, and SPKI certificates were designed for public key distribution. As a result, they contain many features irrelevant to private key management, such as sophisticated naming and authorization languages.

These components add complexity, but they could be ignored or chopped out. Below we will ignore these irrelevancies and focus only on whether these certificate formats have sufficient functionality to meet our requirements.

3.2.1 X.509 Proxy Certificates

Proxy Certificates (or PCs) are X.509 certificates issued under an end-entity certificate or under another proxy certificate [43]. PCs were developed within the Grid Security Infrastructure [5,21] to support single sign-on and delegation of rights within a grid computing environment. Later, PCs were considered for private key management in a fashion similar to our rootholders and certificate servers [38]. Standardization of PCs is being pursued within the IETF PKIX working group [28].

PCs are a simple profile of X.509 certificates – essentially, an extension is added to indicate that a certificate is a PC, and path validation is simplified. Certain of our arguments against PCs thus apply to X.509 certificates in general.

PCs do not possess an authorization language for stating which protocol(s) a key may be used with. PCs *do* allow for the inclusion of arbitrary policy statements, so an appropriate language could be defined.

X.509 certificates do not allow threshold subjects – a certificate can only certify a single public key.

X.509 certificates *do* allow timed and one-time revalidation through CRLs [25] and OCSP responses [37] issued by Indirect CRL Authorities and Designated OCSP Responders (what we have called VAs). However, threshold VAs are not supported. Furthermore, a VA issuing these instruments must have a certificate from the CA who issued the subject certificate. This causes some problems:

- It is less efficient than putting the VA's public key in the subject certificate.
- It complicates revalidation of root certificates.
- It is unclear whether Proxy Issuers are allowed to issue certificates to VAs.

Another problem is that not all application protocols allow CRLs and OCSP responses to be transmitted along with certificate chains.

Proxy certificates also inherit a feature from X.509 certificates that we consider a security risk. Suppose a CA or Proxy Issuer has multiple certificates for the same key, with the same name. X.509 certificate chaining is based on the name and key, so certificate chains containing these certificates could be spliced with each other, allowing a downstream subject to place himself under whichever upstream certificates he finds most convenient: "if an issuer were two PCs with identical names and keys, but different rights this could allow the two PCs to be substituted for each other in path validation and effect the rights of a PC down the chain" [43]. This situation can arise quite naturally as certificates are re-issued over time.

Finally, PCs are encumbered with X.509's legacy baggage, such as ASN.1, OIDs, Distinguished Names, ambiguous key usage bits, multiple bolted-on validation mechanisms, and an excessive number of ways to encode strings, represent time, and identify certificates [22].

3.2.2 PGP Subkeys

PGP subkeys [6] were introduced when PGP was trying to avoid the RSA algorithm (for patent reasons) by allowing a user's "key" to be a DSA signing primary key paired with an Elgamal encryption subkey. With this functionality in place, people began to use short-lived encryption subkeys, to give a measure of *forward secrecy* [4] in the face of key compromise. Signing subkeys are not often used, though the PGP community recognizes the benefits of keeping a primary certification key in a safe environment and certifying subkeys for use in more convenient and hence riskier environments [3,32].

A PGP subkey cannot have subkeys, so PGP subkeys couldn't support a 3-tiered architecture of rootholders, certificate servers, and user devices. PGP has some key flags, but these don't have sufficient granularity to precisely specify which protocols a key can be used with. PGP does not support threshold subjects, or revalidations.

3.2.3 SPKI Certificates

SPKI has inspired our requirements, and has informed much of the thinking in this paper. SPKI certificates almost meet our requirements, but we will point out a few deficiencies.

SPKI's authorization language is sufficient for authorizing keys for use with particular protocols.

SPKI supports threshold subjects: a certificate can certify a k-of-n combination of keys or other threshold subjects. Each key then becomes the root of a separate certificate chain, and these certificate chains have to rejoin at some future point by all certifying the same subject. This is rather complex, since a certificate chain can now contain sub-chains. The current SPKI structure draft [13] doesn't specify how these are handled.

SPKI supports timed and one-time revalidations. However, it does *not* support threshold VAs – instead, *every* VA listed in an SPKI certificate must validate the certificate. To achieve a threshold an issuer could issue certificates that name each k-subset of VAs separately. For a 3-of-5 threshold this works out

to 10 certificates. If the subject of these 10 certificates wanted to issue a certificate that also has a 3-of-5 threshold of VAs, he would have to issue 10 certificates under each of his 10 certificates, for a total of 100 certificates. This becomes unmanageable.

A different approach would be to ignore SPKI's specialized validation instruments, and to just treat VAs as threshold subjects [14]. If *A* is the subject key and *B* through *F* are VA keys, the following expression would require 3 of the 5 VAs to collaborate to issue a certificate: *(2 of A,(3 of B,C,D,E,F))*. Of course, normally VAs *don't* collaborate to isssue a certificate, they simply validate the certificate they belong to. Since threshold subjects cannot sign their own certificate, they would each need to issue another certificate whose only subject is *A*. There are a couple of problems with this:

- *A* would need to sign this certificate as well. If *A* was itself a threshold subject, this could add a fair number of extraneous signatures.
- The validation certificates issued by the VAs can't cover multiple certificates with a single signature, as SPKI's specialized validation instruments can.

One last, minor point is that SPKI uses canonical S-Expressions [40] for encoding certificates. This is a less popular text encoding than XML.

In sum: SPKI's threshold subjects seem overly complex and under-specified. SPKI's VAs do not support thresholds. VAs can instead be treated as threshold subjects, but using certificates as validation instruments is clumsy in a few ways. Nonetheless, the idea of integrating VAs as threshold subjects seems promising, if it could be done more cleanly. We will pursue this further in the cryptoID certificate format.

3.3 The CryptoID Certificate System

The cryptoID system currently comprises a fingerprint format and a certificate format. The fingerprint format is designed for human convenience. The certificate format is designed to be simple for relying parties while providing great flexibility in key management for subjects.

In the next few subsections we give a detailed view of these formats. In 3.3.9 we step back and give the rationales behind certain decisions. An example <*certChain*> in the Appendix may help clarify the text.

3.3.1 CryptoIDs

A fingerprint calculated from a cryptoID root certificate we call a *cryptographic identifier*, or *cryptoID*. A cryptoID is a 100 bit value formatted as four groups of five lower-case base 32 digits. The base 32 alphabet we use consists of the letters 'a' through 'z' except 'l', and the numbers 3 through 9. Below are some example cryptoIDs:

```
dhdkc.9af3q.f8rhk.choae

bfmns.8x95s.ch59b.jtrdo

fynze.i9byx.ow7bc.ybwt9

e84vj.8ag5g.skosq.3hxzo
```

As 100 bits may not be secure against a brute-force search for pre-images on a hash function, we adopt the *hash extension* technique from [1]. The last 96 bits of the cryptoID are taken from a SHA-1 hash of the root certificate. The first bit is reserved, and must be set to zero. The next three bits form a *zero count* and are used to determine the number of *zero bytes* by the formula *1 + zeroCount*. The resulting 1 to 8 zero bytes are prepended to the 96 bits to yield a *check value* from 104 to 160 bits in length. When a root certificate is presented to the relying party, the relying party will hash it using SHA-1, and ensure that the check value is a prefix of the hash value.

When generating a root certificate, the cryptoID creator chooses the number of zero bytes he desires. The creator then includes trial *modifier* values in the certificate and hashes the result until a modifier which yields the requisite number of zero bytes is discovered.

Essentially, we're boosting the security level of the fingerprint by requiring an attacker to find a hash collision on the run-length-encoded zero bytes plus the 96 bits. The downside of this is that the creator of the cryptoID must find a hash collision on the zero bytes as well. Thus, the creator of the cryptoID must weigh his security requirements against the inconvenience of a lengthy one-time computation.

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 naturally move to higher levels.

3.3.2 XML Encoding

We use XML for our certificate format. There will be a canonical form for this XML, and whenever a <*certChain*> XML element (see 3.3.4) is used within a non-XML protocol (such as TLS or S/MIME [29]) it must be in canonical form.

Canonicalization is necessary since some elements are used as inputs to hash functions, and it also simplifies parsing. However, when a <*certChain*> is used within an XML document, it does *not* need to be in canonical form, as long as it is canonicalized before hashing. We won't define the canonical form in this document. However, it simply consists of using a default namespace, using tabs for indenting, and not breaking up long lines of text, plus a few other things.

For defining XML structures within this document, we use a shorthand similar to regular expressions, where:

- "?" denotes zero or one occurrence
- "+" denotes one or more occurrence
- "*" denotes zero or more occurrences
- x|y denotes either x or y

For encoding binary data or large numbers, we use base64 encoding in the same fashion as XML-Signature [11].

3.3.3 Application Protocols

The <*certChain*> element (see 3.3.4) is presented to a relying party by the owner of a cryptoID, and serves to certify a particular end-entity key.

We would like to retrofit protocols such as TLS, S/MIME, and IPsec [26] to support cryptoID certificate chains. A <*certChain*> can be transmitted in any of these protocols by sending it instead of an X.509 chain. In addition, care must be taken to ensure that any use of the private key commits to the <*certChain*>.

Some examples:

- In TLS [9], we can use the *cert_type* extension from [34] to indicate that a <*certChain*> will be carried in the TLS *Certificate* messages.
- In S/MIME [29], we can add a <*certChain*> as a signed attribute in a *SignerInfo* structure. Within a *RecipientIdentifier*, we can use *subjectKeyIdentifier* to carry a *chainID* (see 3.3.4).
- In IPsec's IKEv2 [31], we can add a *Certificate Encoding* for carrying a <*certChain*> within a *Certificate Payload*.
- In XML-Signature and XML-Encryption [11,12], we can add a <*certChain*> inside a *dsig:KeyInfo*.

3.3.4 The <certChain> Element

A <*certChain*> has a few attributes and child elements:

```
<certChain chainID cryptoID>
    <certs>
    <publicKeys>
    <signatures>?
</certChain>
```

The *chainID* attribute gives a SHA-1 hash of the certificates within the <*certs*> element. This value uniquely identifies the certificate chain. The *cryptoID* attribute gives the relevant cryptoID.

The <*certs*> element contains a list of certificates. Each certificate binds a *key expression* to a set of authorized protocols. A key expression is like a threshold subject in SPKI: the variables of the expression correspond to key hashes, and these variables are linked by threshold connectives. Each variable evaluates to true if the corresponding key produces a *certification signature* on the next certificate in the chain, or a *validation signature* on the current certificate. The first variable in the last key expression is the end-entity key, which always evaluates to true. Each key expression must evaluate to true for the chain to be valid.

The <*publicKeys*> element lists public keys that match the hashes in the key expressions.

The <*signatures*> element lists certification and validation signatures that satisfy the key expressions.

3.3.5 The <certs> Element

```
<certs>
  <cert>+
</certs>

<cert certID>
  <keyExpression>
  <notAfter>?
  <protocols>?
  <modifier zeroCount>?
</cert>
```

The last certificate in the chain with a <*modifier*> value is known as the *root certificate.* To construct a cryptoID for the chain, a SHA-1 hash is calculated starting with the start tag of the first certificate, and ending with the <*modifier*> start tag; then zeros are added to pad the value to 64 bytes, and the modifier data is hashed. This ensures that trying a modifier only requires a single application of the SHA-1 compression function.

In many cases the first certificate will be the root certificate. In other cases it is desirable to have a cryptoID-independent portion of the chain before the root certificate introduces the cryptoID-dependent portion. The cryptoID-independent portion allows a root key to receive certification or validation signatures independent of any particular cryptoID, instead of needing separate signatures for each cryptoID it plays a role in.

Each cert must be given a *certID*, which is a random, base64-encoded 20-byte value that can be used to uniquely identify the certificate in key management protocols. Each cert may also have a <*notAfter*> expiration time. The <*keyExpression*> and <*protocols*> elements are discussed below.

Each cert is known by its index within the chain. The first <*cert*> is 0, the next is 1, and so on to a maximum value of 9.

3.3.5.1 The <keyExpression> Element

Each <*cert*> has a <*keyExpression*>, containing an *expr* attribute and a list of <*keyHash*> elements.

```
<keyExpression expr>
    <keyHash>*
</keyExpression>
```

Each <*keyHash*> in the chain binds a new variable. The first variable is "A", the next "B", and so on, to a maximum of "Z". Each <*keyHash*> takes the SHA-1 hash of a <*publicKey*> (see 3.3.6) as its content.

The *expr* attribute contains the key expression proper. Each *expr* can refer to variables bound in the current certificate or any previous certificate. In other words, once a variable is bound, it remains bound for the rest of the certificate chain.

The *expr* attribute states which combinations of keys can jointly exercise the certificate's power. Any key in a key expression can issue a certification signature on the next certificate in the chain, or a validation signature on the current certificate. A certification signature is the key's way of assenting to a particular exercise of power. A validation signature is the key's way of expressing trust in the other keys in the key expression, and delegating its decision-making powers to them.

Below is the key expression from the first <*cert*> of the <*certChain*> listed in the Appendix:

```
<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>
```

"A", "B", and "C" are CA keys. "D" and "E" are VA keys. This distinction is not encoded in the key expression, but is a matter of operational behavior. Below is the key expression from the next and last certificate in the same chain:

```
<keyExpression expr="(F and (D or E))">
   <keyHash>FjhqXDC+MTkhP8TjUO0AduGqNOo=</keyHash>
</keyExpression>
```

This expression draws upon the VAs "D" and "E" which were previously bound, and introduces the end-entity key "F". In the last certificate in a chain, the first key in the expression is always the end-entity key, and the other keys are restricted to producing validation signatures. Since "D" and "E" are listed in key expressions in two certificates, a single signature from either of them could be used to validate both certificates.

Below is pseudo-BNF for the *expr* string. The last key expression in the chain must match the *lastExpr*:

```
expr ::= andExpr | orExpr | threshExpr | var
lastExpr ::= "(" var " and " expr ")" | var
andExpr ::= "(" expr " and " expr ")"
orExpr ::= "(" expr " or " expr ")"
threshExpr ::= "(" number " of " exprList ")"
exprList ::= expr | expr "," exprList
number ::= "1"-"9"
var ::= "A"-"Z"
```

3.3.5.2 The <protocols> Element

Each <*cert*> may have a <*protocols*> element, listing the protocols which the key expression can issue certificates for, or which the end-entity key can be used with.

```
<protocols>
    <protocol>+
</protocols>
```

If <*protocols*> is omitted, the <*keyExpression*> is usable with all protocols. Each <*protocol*> element consists of a URI. For example:

- *http://trevp.net/instant-messaging*
- *http://trevp.net/voice*

These values could be used for voice or instant-messaging authentication.

The <*protocols*> element in each <*cert*> must be a subset of the previous <*cert*>'s <*protocols*>. A protocol is considered a subset of another if it contains the other protocol as a prefix. To make comparing lists of protocols easy, we require procotols to appear in lexicographic order.

3.3.6 The <publicKeys> Element

The <*publicKeys*> element contains a list of keys:

```
<publicKeys keys>
    <publicKey>+
</publicKeys>
```

Each <*publicKey*> will be qualified with a default namespace which determines its type, and thus its contents. For example:

```
<publicKey xlmns="http://trevp.net/rsa">
    <n>8b1hBlWQ...</n>
    <e>Aw==</e>
</publicKey>
```

A more precise type, such as *http://trevp.net/rsa-pkcs1-sha1*, could limit the key for use with a particular algorithm.

The *keys* attribute is a list that matches each key with a corresponding <*keyHash*>. Not all <*keyHash*> elements need to have a corresponding <*publicKey*> - only keys that have produced signatures, plus the end-entity key, need to be listed in <*publicKeys*>.

3.3.7 The <signatures> Element

The <*signatures*> element contains a list of signatures:

```
<signatures>
    <signature chaff key listCA? listVA?
     notAfter? sigAlgo>+
</signatures>
```

The *chaff* attribute is a random value added by the signer to prevent collision attacks.

The *key* attribute identifies the key performing this signature.

The *listCA* attribute lists every cert for which this key, in this signature, is performing the function of a CA. The *listVA* attribute lists every cert for which this key, in this signature, is performing the function of a VA. These attributes can be present simultaneously. For every cert in *listCA* or *listVA*, the variable matching the signer's key is set to *true.*

The *notAfter* attribute gives the date and time when this signature expires, using the *dateTime* UTC format from XML-Schema [2]. If *notAfter* is not present, the signature never expires.

The *sigAlgo* attribute specifies the asymmetric algorithm used for the signature.

The <*signature*> content is a base64-encoded signature value. The signature is calculated by hashing a concatenation of:

- The <*signature*> start tag, including all its attributes.
- Some number of <*cert*> elements, starting with the first <*cert*>'s start tag, and ending with the end tag of the *Xth* <*cert*>, where *X* is the last <*cert*> covered by this signature.

  To calculate *X*, take the largest number *V* in the *listCA* list and the largest number *C* in the *listCA* list. Add one to *C*. Set *X = max(V,C).*

This construction serves two purposes:

- Hashing the <*signature*> start tag, with its chaff value, before the certificates, means that the signature will be calculated on a hash result that can't be controlled by an attacker.
- Hashing all the certificates from the first to the *Xth* means that all important context is covered by the signature, so there's no possibility of certificate chain manipulations like those that X.509 is susceptible to.

Below is an example <*signatures*> from the Appendix that would satisfy the previous example key expressions:

```
<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>
```

3.3.8 The <certChain> Validation Algorithm

To validate a <*certChain*> requires the following steps:

1) Check that <*certs*> yields a cryptoID consistent with the root <*cert*>'s *zeroCount* attribute, and equal to the *cryptoID* attribute.

2) Check that the *chainID* attribute equals the SHA-1 hash of all the certificates within the <*certs*> element.

3) Check that every <*publicKey*> matches its <*keyHash*>.

4) Check that each certificate's <*protocols*> and <*notAfter*> values are a subset of the previous.

5) Check that any certificate and signature *notAfter* values are in the future.

6) Check that the signatures verify correctly.

7) Set each variable in each key expression to *false.*

8) For each *listCA* and *listVA* number in each signature, set the variable for the signer's key in the relevant key expression to *true.*

9) Set the first variable in the last key expression to *true.*

10) Check that each key expression evaluates to *true.*

11) Check that each cert except the last has at least one key performing a CA signature which contributes to the result of the key expression. By "contributing to the result" we mean that if the key expression is viewed as a parse tree, the intermediate nodes between the key variable (leaf) and the result (root) all evaluate to true. Thus in the key expression "((A and B) or C)", if only "A" and "C" are true, validation will be successful only if "C" performs a CA signature, since "A" does not contribute to the result.

3.3.9 Rationales

We end this section by trying to justify some design decisions.

3.3.9.1 CAs and VAs

One innovation in cryptoID certificates is to integrate the roles of CAs and VAs. A <*keyExpression*> does not assign any key a particular role – any key in the expression can issue *either* certification *or* validation signatures.

We view validation signatures as an optimization: if one key trusts another, the first key could echo all the certification signatures performed by the other. A validation signature simply lets the first key express this trust, so the first key doesn't have to be contacted each time the other key signs a certificate.

3.3.9.2 Key Hashes

We could have placed keys directly inside certificates, instead of using <*keyHash*> elements. One advantage of referring to keys through their hashes is that unused keys can be omitted. Another rationale is that since the certificates have to be hashed for every signature, we wanted to keep the certificates themselves small.

3.3.9.3 Hashing of Preceding Certificates

The signature on any <*cert*> covers all preceding certificates. This makes the certificate format simple, since each <*cert*> doesn't have to identity its issuer, reiterate the cryptoID it belongs under, or re-bind variables. Also, it prevents any splicing of one certificate chain with another.

3.3.9.4 Modifier Placement

The modifier is placed at the end of a <*cert*>. This allows the cryptoID creator to calculate the SHA-1 hash of the root <*cert*> except for the last block, and then only hash the last block for each trial modifier. If the modifier was placed earlier, the legitimate cryptoID creator would be slowed down, while an attacker could repurpose other elements for use as modifiers, and still perform efficient searching.

3.3.9.5 CAs vs. EEs

A cryptoID certificate can be used directly in a protocol, or it can issue end-entity certificates for that protocol. Some certificate systems allow each certificate only one of these uses. However, this distinction is not enforceable: with a CA certificate you can

issue yourself an EE certificate, and with an EE certificate you can give away the private key or else proxy requests for other people.

4. CONCLUSION

Conventionally, certificates have been used for key distribution. We argued that they are unsuited for this – they can only delegate in limited ways, and the resulting structures are too complex. Instead, users should exchange small fingerprints as if they were addresses. These fingerprints would be easy for users to handle and easy for software to support.

Certificates work better when used for private key management – here there's no need to delegate over namespaces or authorizations, and there's no need for structures more complex than chains. The only function of key management certificates is to allow multiple subkeys under a single fingerprint, and to limit the damage done if subkeys are compromised. This allows the user to have a root key which is managed in a highly secure fashion, while subkeys are used for day-to-day business.

We then argued that current certificate systems are too complicated for such a simple use, and lack certain desirable features, so we designed a system focused on this case. The result is the cryptoID – a small, multipurpose, long-lived identifer which places key distribution where it should be – in the hands of users.

5. ACKNOWLEDGMENTS

Thanks to the anonymous reviewers and the participants at NSPW 2003 (in particular Bob Blakley) for their advice.

6. AVAILABILITY

An implementation of the cryptoID system is available at http://trevp.net/cryptoID/.

7. REFERENCES

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

[2] P.V. Biron and A. Malhotra. W3C Recommendation: XML Schema Part 2: Datatypes, May 2001. http://www.w3.org/TR/xmlschema-2/

[3] I. Brown. RE: OpenPGP Sub Keys (Was: key flag for authentication). *OpenPGP Mailing List*, June 2003. http://www.imc.org/ietf-openpgp/mail-archive/msg05207.html

[4] I. Brown, A. Back, and B. Laurie. Internet-Draft: Forward Secrecy Extensions for OpenPGP, April 2002. http://www.cs.ucl.ac.uk/staff/I.Brown/draft-brown-pgp-pfs-03.txt

[5] R. Butler, D. Engert, I. Foster, C. Kesselman and S. Tuecke. A National-Scale Authentication Infrastructure. *IEEE Computer*, 33(12), 2000. http://www.globus.org/documentation/incoming/butler.pdf

[6] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. RFC 2440: OpenPGP Message Format, November 1998 http://www.ietf.org/rfc/rfc2440.txt

[7] T. Close. What Does the 'y' Refer to?, July 2003. http://www.waterken.com/dev/YURL/Definition/

[8] M. Cooper et. al. Internet Draft: Internet X.509 Public Key Infrastructure: Certification Path Building, February 2003. http://www.ietf.org/internet-drafts/draft-ietf-pkix-certpathbuild-00.txt

[9] T. Dierks and C. Allen. RFC 2246: The TLS Protocol Version 1.0, January 1999 http://www.ietf.org/rfc/rfc2246.txt

[10] W. Diffie and M.E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory, 22*, 1976. http://citeseer.nj.nec.com/340126.html

[11] D. Eastlake, J. Reagle, and D. Solo. RFC 3075: XML-Signature Syntax and Processing, March 2001. http://www.ietf.org/rfc/rfc3075.txt

[12] D. Eastlake and J. Reagle. W3C Recommendation: XML Encryption Syntax and Processing, December 2002. http://www.w3.org/TR/xmlenc-core/

[13] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. Internet Draft: Simple Public Key Certificate, July 1999. http://world.std.com/~cme/spki.txt

[14] Ibid., 4.5.3(4)

[15] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. RFC 2693: SPKI Certificate Theory, September 1999. http://www.ietf.org/rfc/rfc2693.txt

[16] Ibid., 1.1.

[17] Ibid., 5.3.

[18] Ibid., 5.5.

[19] Ibid., 7.

[20] Ibid., 7.6.

[21] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. *Proceedings of the 5th ACM Conference on Computer and Communications Security*, 1998. ftp://ftp.globus.org/pub/globus/papers/security.pdf

[22] P. Gutmann. X.509 Style Guide, October 2000. http://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.txt

[23] P. Gutmann. Everything you Never Wanted to Know about PKI but were Forced to Find Out. http://www.cs.auckland.ac.nz/~pgut001/pubs/pkitutorial.pdf

[24] P. Hallam-Baker. W3C Working Draft: XML Key Management Specification Version 2.0, April 2003. http://www.w3.org/TR/xkms2/

[25] R. Housley, W. Polk, W. Ford, and D. Solo. RFC 3280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, April 2002 http://www.ietf.org/rfc/rfc3280.txt

[26] IETF IPsec Working Group (ipsec) http://www.ietf.org/html.charters/ipsec-charter.html

[27] IETF OpenPGP Working Group (openpgp). http://www.ietf.org/html.charters/openpgp-charter.html

[28] IETF Public Key Infrastructure Working Group (pkix). http://www.ietf.org/html.charters/pkix-charter.html

[29] IETF S/MIME Working Group (smime) http://www.ietf.org/html.charters/smime-charter.html

[30] IETF TLS Working Group (tls) http://www.ietf.org/html.charters/tls-charter.html

[31] C. Kaufman. Internet Draft: Internet Key Exchange (IKEv2) Protocol, August 2003. http://www.ietf.org/internet-drafts/draft-ietf-ipsec-ikev2-10.txt

[32] W. Koch. RE: OpenPGP Sub Keys. *OpenPGP Mailing List*, June 2003. http://www.imc.org/ietf-openpgp/mail-archive/msg05209.html

[33] A. Malpani, R. Housley, and T. Freeman. Internet Draft: Simple Certificate Validation Protocol (SCVP), June 2003. http://www.ietf.org/internet-drafts/draft-ietf-pkix-scvp-12.txt

[34] N. Mavroyanopoulos. Internet Draft: Using OpenPGP keys for TLS authentication, April 2003. http://www.ietf.org/internet-drafts/draft-ietf-tls-openpgp-keys-03.txt

[35] M. Miller. Lambda For Humans – The Pet Name Markup Language. http://www.erights.org/elib/capability/pnml.html

[36] M. Miller, C. Morningstar and B. Frantz. Capability-based Financial Instruments. *Proceedings of Financial Cryptography*, 2000. http://www.erights.org/elib/capability/ode/index.html

[37] M. Myers, R. Ankney, A. Malpani, S. Galperin and C. Adams. RFC 2560: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol, June 1999. http://www.ietf.org/rfc/rfc2560.txt

[38] J. Novotny, S. Tuecke, and V. Welch. An Online Credentials Repository for the Grid: MyProxy. *Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10)*, August 2001. http://www.globus.org/research/papers/myproxy.pdf

[39] D. Pinkas and R. Housley. RFC 3379: Delegated Path Validation and Delegated Path Discovery Protocol Requirements, September 2002. http://www.ietf.org/rfc/rfc3379.txt

[40] R. Rivest. Internet-Draft: S-Expressions, May 1997. http://theory.lcs.mit.edu/~rivest/sexp.txt

[41] R. Rivest and B. Lampson. SDSI – A Simple Distributed Security Infrastructure. *Presented at Crypto '96 Rump Session*, 1996. http://theory.lcs.mit.edu/~cis/sdsi.html

[42] D. Taylor, T. Wu, N. Mavroyanopoulos, and T. Perrin. Internet-Draft: Using SRP for TLS Authentication, June 2003. http://www.ietf.org/internet-drafts/draft-ietf-tls-srp-05.txt

[43] S. Tuecke et. al. Internet Draft: Internet X.509 Public Key Infrastructure Proxy Certificate Profile, August 2003. http://www.ietf.org/internet-drafts/draft-ietf-pkix-proxy-08.txt

[44] A. Whitten and J.D. Tygar. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. *8th Usenix Security Symposium*, 1999. http://www-2.cs.cmu.edu/~alma/johnny.pdf

[45] Ibid., 4.4.

[46] Ibid., 5.3. *"Deciding whether to trust keys from the keyserver"*.

[47] Ibid., 5.3. "*Getting other people's public keys*".

APPENDIX - 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>
```