Enhanced certificate transparency (how Johnny could encrypt)

Mark D. Ryan
School of Computer Science
University of Birmingham, UK
http://www.cs.bham.ac.uk/~mdr

***Abstract*—The “certificate authority” model for authenticating public keys of websites has been attacked in recent years, and several proposals have been made to reinforce it. We develop and extend “certificate transparency”, a proposal in this direction, so that it efficiently handles certificate revocation. We show how this extension can be used to build a secure end-to-end email or messaging system using PKI with no requirement to trust certificate authorities, or to rely on complex peer-to-peer key-signing arrangements such as PGP. We believe this finally makes end-to-end encrypted email as usable as encrypted web browsing is today, addressing the concerns of a classic paper explaining the difficulties users face in encrypting emails (“Why Johnny can’t encrypt”, 1999). Underlying these ideas is a new attacker model appropriate for cloud computing, which we call “malicious-but-cautious”.**

1 Introduction

1.1 Background and motivation

Public-key cryptography relies on entities being able to obtain authentic copies of other entities’ public keys. For example, suppose a user wishes to log in to their bank account through their web browser. The web session will be secured by the public key of the bank. If the user’s web browser accepts the wrong public key for the bank, then the traffic (including login credentials) can be intercepted and manipulated by an attacker.

In order to avoid such attacks, *certificate authorities* (CAs) are used assure an entity about the public key of another one. In the example given, the browser is presented with a public key certificate for the bank, which is intended to be unforgeable evidence that the given public key is the correct one for the bank. The certificate is digitally signed by a CA. The browser is pre-configured to accept certificates from certain known CAs. A typical installation of Firefox has about 100 CAs in its database.

Aside from cryptography issues [41], [29], [28], there are two big problems with the CA model. Firstly, CAs must be assumed to be trustworthy. If a CA is dishonest or compromised, it may issue certificates asserting the authenticity of fake keys; those keys could be created by an attacker or by the CA itself. Unfortunately, the assumption of honesty does not scale up very well. As already mentioned, a browser typically has hundreds of CAs registered in it, and the user cannot be expected to have evaluated the trustworthiness of all of them. This fact has been exploited by attackers. If an attacker manages to insert a malicious CA into the user’s browser, the attacker can get the browser to accept fake keys for standard services (such as bank web sites and webmail sites). Then the attacker can intercept and manipulate the user’s traffic with those sites. Many attacks based on these ideas have been reported [12], [20], [16], [44], [46], [35]. In 2011, two CAs were compromised: Comodo [26] and DigiNotar [7]. In both cases, certificates for high-profile sites were illegitimately obtained, and in the second case, reportedly used in an MITM attack [8].

A second problem with the CA model is key revocation. If a certificate owner loses control of its private key, it needs to revoke the certificate before its expiration date. Currently, web browsers attempt to check revocation on the fly: the browser queries the CA to verify that a certificate hasn’t been revoked. Unfortunately, that solution doesn’t work well; it poses a large burden on CAs to respond to such requests; it can be defeated by attackers that block such requests; and it has privacy implications for web users.

Several interesting solutions have been proposed to address these problems. (For a good survey, see [13].) Certificate pinning addresses the problem of untrustworthy CAs, by restricting in the client browser parameters concerning the set of CAs that are considered entitled to certify the key for a given domain [1], [21]. Crowd-sourcing techniques have been proposed in order to detect untrustworthy CAs, by enabling a browser to obtain warnings if the certificates it is offered are out of line with those that other people are being offered [11], [17], [14], [4]. In another direction, certificate transparency [6] is an approach which aims to prevent certificate authorities from issuing public key certificates for a domain without being visible to the owner of the domain. The core idea is that a public append-only log is maintained, showing all the certificates that have been issued. A certificate is accepted only if it is accompanied by a proof that it has been inserted into the log.

Solutions for revocation management have also been proposed; they mostly involve periodically pushing revocation lists to browsers, in order to remove the need for on-the-fly revocation checking [18], [2]. However, this solution creates a window during which the browser’s revocation lists are out of date until the next push. Revocation transparency [5] is

an extension of certificate transparency that aims to deal with revocation, although (as we later explain) we believe it does not scale up, since it requires space and time that is linear in the number of revocations.

1.2 Extending certificate transparency

Certificate transparency solves the problem that CAs are required to be trusted. It uses public logs and optionally gossip protocols to ensure that CAs leave persistent evidence of all the certificates they issue. In that way, the activities of a CA are visible ("transparent") to its users and to observers. Users accept a certificate only if it is accompanied by a proof that it is included in the log. The proofs are short and efficiently verifiable by browsers. Even if there are 10^9 certificates, the proofs amount to 1KB or 2KB of data.

Unfortunately, certificate transparency does not handle revocation efficiently. The core proposal in the IETF draft [6] does not specify any revocation mechanism. An informal proposal for handling revocation exists [5], but adopting it has the side-effect of dramatically reducing the efficiency of certificate transparency. Roughly speaking, the size of proofs goes from 1KB or 2KB to tens or hundreds of GB.

We extend certificate transparency to handle revocation efficiently. In our extension, proofs that a key is *current* (i.e., issued and not revoked) are as efficient as proofs of issuance in certificate transparency. Proofs of absence (i.e., proofs that a CA has not issued any certificates for a subject) are also as efficient. Thus, all the proofs that browsers request are efficient in our extension.

1.3 End-to-end encrypted mail

Public-key cryptography was invented to allow users to send encrypted mail[1]; nevertheless, 35 years later, in practice it is rather hard for users to encrypt their mail in a systematic way. "Why Johnny can't encrypt" is a 1999 classic paper [23] explaining why PGP encryption for email has failed to take off. This failure of adoption of encryption for mail is in marked contrast with the encryption for the web, where encrypted browsing is routinely done by billions of users each day. Numerous efforts to improve this situation have been made (a brief review is in §2.4), but none of them simultaneously satisfy the requirements of **usability** (users should not be required to understand anything about keys, or to take any special actions) and **security** (encryption should be end-to-end, and there should be no trusted parties).

Certificate transparency was developed for web certificates, but we demonstrate in this paper that, once extended to handle revocation efficiently, it has the ability to solve the problem of end-to-end encrypted email too. As mentioned, the core property of certificate transparency is that it allows CAs to be untrusted. By using (our extension of) certificate transparency as a foundation, we detail a method in which an untrusted provider can act both as a CA and as a provider of the email service. This allows users to send encrypted mail without having to understand anything about keys or certificates, and without having to rely on any trusted parties.

[1] Indeed, the first line of the 1978 RSA paper is: *The era of "electronic mail" may soon be upon us; we must ensure that [...] messages are private.*

1.4 Our contribution

We develop and extend the idea of certificate transparency, and we apply it to email encryption. In particular,

- We rework certificate transparency so that it properly handles revocation, in space/time which is logarithmic in the number of revocations.
- We show how it can be used to build a usable and secure email or messaging service using PKI with no trusted parties.
- We develop a new attacker model appropriate for cloud computing, which we call "malicious-but-cautious".

Structure of paper: In section 2, we review some background material on which the paper relies. Section 3 details our extension to certificate transparency, which we call *certificate issuance and revocation transparency* (CIRT) to emphasise that it efficiently handles certificate revocation as well as issuance. In section 4, we describe the application of CIRT to email, and show that it enables end-to-end encrypted email without requiring any trusted parties (such as CAs), and without requiring any additional understanding or effort from users. Discussion of our attacker model, called the "malicious but cautious" attacker, is made in section 5.

2 Background

We review some of the background material on which the paper relies.

2.1 Merkle trees

A Merkle tree is a tree in which every node is labelled with the hash of the labels of its children nodes, and possibly some other values. Suppose a node has n children labelled with hash values $v_1, \ldots, v_n$, and has data d. Then the hash value label of the node is the hash of $v_1, \ldots, v_n, d$. Merkle trees allow efficient proofs that they contain certain data. To prove that a certain data item d is part of a Merkle tree requires an amount of data proportional to the log of the number of nodes of the tree. (This contrasts with hash lists, where the amount is proportional to the number of nodes.)

Example: Figure 1 shows a Merkle tree containing data items $c_1, \ldots, c_6$ stored at the leaf nodes (in this tree, there are no data items stored at non-leaf nodes). Figure 2 shows a larger Merkle tree containing data items $c_1, \ldots, c_{32}$ (again in this case stored only at leaves). To demonstrate that c_{11} is present in the tree, it is sufficient to provide the additional data $c_{12}, h_5, h_{14}, h_{16}, h_{20}$, i.e. one data item per layer of the tree.

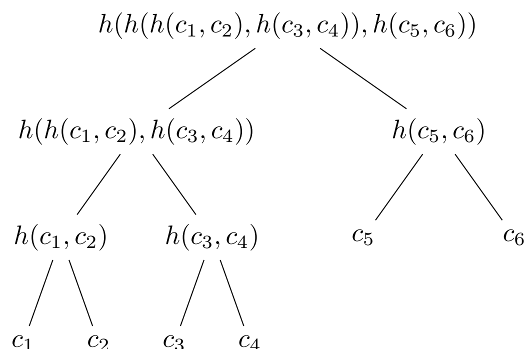

Fig. 1. A Merkle tree containing items $c_1, \ldots, c_6$.

The recipient of this data can then verify the correctness of the root hash h_{21}. Proving that one Merkle tree extends another can also be done in logarithmic space and time, by providing at most one hash value per layer. For example, to demonstrate that the tree of Figure 2 is an extension of the one in Figure 1, it is sufficient to provide the data h_4, h_{17}, h_{20}.

2.2 Certificate transparency

Certificate transparency [6], [9], [40] is a technique invented by Google that aims to prevent TLS [33], [47], [32] certificate authorities from issuing public key certificates for a domain without being visible to the owner of the domain. It is aimed at website certificates, and the technology is being built into Google Chrome.

The core idea is that a public log is maintained, showing all the certificates that have been issued. The log is append-only. Anyone can append a certificate to the log. Auditors can obtain two types of proofs: (a) a proof that the log contains a given certificate, and (b) a proof that a snapshot of the log is an extension of another snapshot (i.e., only appends have taken place between the two snapshot).

Abstractly, we may consider a certificate as a signed pair $(subj, pk_{subj})$, asserting that a subject $subj$'s public key is pk_{subj}. In certificate transparency, the CA's database of certificates is maintained as a Merkle tree in which these pairs are stored left-to-right in chronological order at the leaves of the tree (Figures 1 and 2). Items are added chronologically, by extending the tree to the right. A certificate is accepted by a browser only if it is accompanied by a proof that the subject-key pair has been inserted into the log. Observers can check that the log is maintained as append-only. To perform such a check, the observer submits to the CA the hash value of the log at two different times. The CA returns a proof that the log corresponding to the later hash value is an extension of the log at the earlier time. The properties of Merkle trees ensure that insertion into the log, and both proofs (the proof of presence in the log, and the proof of extension of the log), can be done in time/space $O(\log n)$.

Linearity: It is vital that the log is a single linear record. If the log maintainer can create different versions of the log to show to different users, the security is lost. Linearity is maintained in two ways. Firstly, whenever a user interacts with the log, it requests proof that the current snapshot is an extension of the previously cached snapshot. This is best done before authentication, to avoid the possibility that a version specially constructed for a particular user is being used. Second, gossip protocols [15] can be used to disseminate values of the log. This means that users of the log (that is, client browsers) need to have a way to exchange with other users the value of the hash of the log that they have received. At any time, a user can request proof that the snapshot currently offered by the log is an extension of a previous snapshot received through direct communication with other users.

2.2.1 Revocation transparency: In certificate transparency, one can prove that a certificate is in the log, but there is no notion of whether it is still current. Revocation transparency [5] is an extension of certificate transparency that aims to deal with revocation. Two alternative methods for revocation transparency are proposed. The first method stores revocations in a data structure called a sparse Merkle tree, which is a Merkle tree in which most of the leaves are zero. A path in this tree has length 256, and represents the hash of a certificate. The path ends in a 1 or 0 leaf according to whether the certificate is revoked or not. The tree is thus a binary tree with 2^{256} leaves, but because it is sparse, these leaves do not have to be stored individually. To revoke a certificate, one alters the sparse Merkle tree so that the relevant path terminates in a 1, and one enters a record of this action in the certificate-transparency append-only log. Unfortunately, checking whether a certificate has been revoked is inefficient. One method is to track the revocations by a separate mechanism, an action which is linear in the number of revocations, which in turn can be assumed proportional to the number of issued certificates. Alternatively, one can check the entire certificate-transparency log for revocation records, but this is again linear in the number of issued certificates. In §3.2, we show that proofs requiring linear space and time require data sizes measured in tens or hundreds of gigabytes, which makes them impractical.

As mentioned, the sparse Merkle tree is one of the proposed data structures for revocation transparency. The other one is a sorted list organised as a search tree. It is also used in conjunction with the certificate-transparency log. But, similarly to the sparse Merkle tree, the ideas only address how revocations should be stored. Checking revocations remains linear in the number of issued certificates.

2.3 Other approaches to handling certificates securely

There are many other proposals for ensuring the authenticity of public key certificates. Early ones are based on croud-sourcing, where a user's assurance that a certificate is genuine

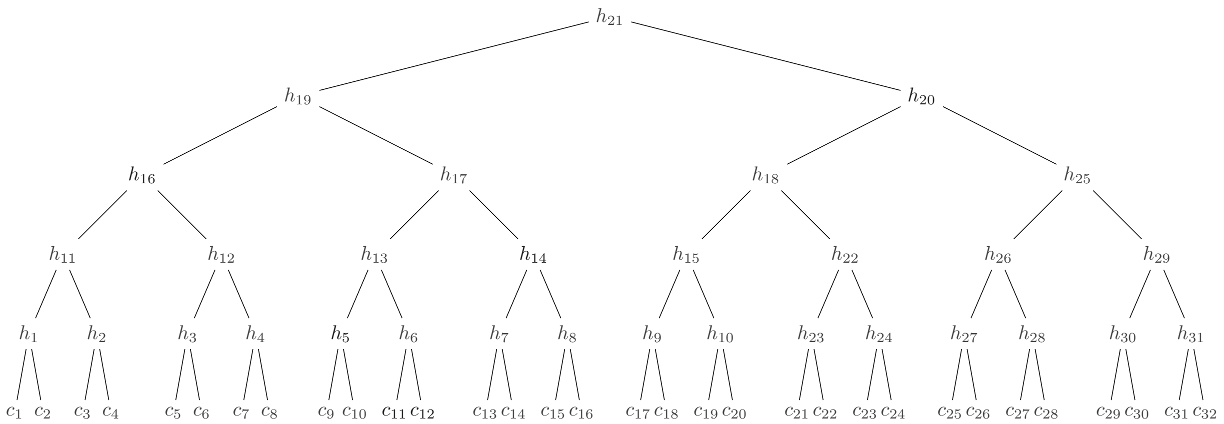

Fig. 2. A Merkle tree containing items $c_1, \ldots, c_{32}$. To demonstrate that c_{11} is present in the tree, it is sufficient to provide the additional data $c_{12}, h_5, h_{14}, h_{16}, h_{20}$. To demonstrate that this tree is an extension of the one in the previous figure, it is sufficient to provide the data h_4, h_{17}, h_{20}.

is increased if other users have received the same certificate. Proposals in this vein include the SSL Observatory [25]; Certificate Patrol [24]; Perspectives [48]; DoubleCheck [30]; CertLock [45]; Covergence [42]; and TACK (2012) [43]. There are also approaches based on using DNS, such as DANE [36]; and CAge (13') [38].

Sovereign Keys [34] is, like certificate transparency, based on the idea of a public log. Another recent proposal that mixes several ideas and also relies heavily on public logs is Accountable Key Infrastructure (AKI) [39].

2.4 End-to-end email encryption

As well as being useful to authenticate public keys for organisations and web sites, public-key certificates can be used for individuals, allowing end-to-end encrypted email. If Alice wishes to send an encrypted email to Bob, she needs to obtain an authentic copy of Bob's public key. There are two main standards in use for public key encryption of email, called S/MIME[2] and PGP[3]. They both require the user's client software to maintain the user's private key, and the public keys of the people she exchanges email with. The main conceptual difference between S/MIME and PGP is the way in which a user verifies that he has an authentic copy of another user's public key. In S/MIME, public keys come with a certificate from a CA. If Bob is an employee of a large corporation such as Boeing, his company may act as a certificate authority for his email public key. But if Bob's email address is from a smaller organisation or is not a company address, there is no natural certificate authority. If there is one, then as previously mentioned, users have to assume it is honest, which may not be a reasonable assumption. For these reasons, S/MIME really works only in a large corporation environment, where the corporation can act as a CA for all its employees. It is natural for both employees and external users that correspond with employees to trust the corporation for email related to its business. S/MIME works less well for small organisations, because they may not wish to take on the complexities of being a CA.

PGP is targeted at individual email users rather than corporate users, and aims to avoid the requirement of "authorities" that certify public keys. This recognises that, in the case of individuals, there are no entities that can fulfil the requirements of being a CA (namely: well-known, trusted by all users, and free to use). To solve this, PGP spreads the certifying role across a set of users, each of whom are somewhat trusted and somewhat known to the sender and receiver, with the expectation that, taken together, this comprises enough evidence for the authenticity of the public key. By signing each other's keys in a peer-to-peer fashion, PGP users create a "web of trust" that works not because of some highly trusted pillars like CAs, but because all the users support the trust web in a small way.

[2]S/MIME stands for *Secure MIME*, and was designed in 1995 as an extension of the MIME format. MIME stands for *Multipurpose Internet Mail Extensions* and is the standard for email attachments. S/MIME version 3 (1999) is standardised by IETF.

[3]The first version of PGP was designed in 1991. The name, *Pretty Good Privacy*, is intended to be humorously ironic. OpenPGP, created in 1997, is an open specification being standardised by IETF.

2.4.1 Inhibitors to take-up of email encryption:

In spite of support on all major client software and significant efforts at supporting take-up, very few people use encrypted mail. Yet, there are substantial motivations, including compliance requirements as well as confidentiality requirements. End-to-end encrypted mail seems to have a dedicated following among a small number of people in very specific sectors.

"Why Johnny can't encrypt" is a 1999 classic paper [23] explaining why PGP encryption for email has failed to take off. Other papers have developed the explanation further. The reasons encrypted email is not routinely used are:

- It is too complicated for users to understand the model. S/MIME is presented to users in a gobbledygook way, asking them to understand public and private keys, key servers, certificates, certificate authorities, etc. Most users

don't want to have to spend time learning this sort of stuff. The pain outweighs the gain.

- S/MIME assumes a hierarchical certificate-authority system for certifying keys which is expensive and cumbersome even for companies, and it appears to be prohibitive for SMEs and individuals. PGP is aimed more at individuals, having a peer-to-peer certifying arrangement, but this also has proved impossible for any but the most determined users to master.
- Even when set up on one platform (e.g., work desktop), the set-up has to be done again on other platforms (laptop, phone) and is different each time. Again, users have to copy keys around between devices, and the set-up is different in different contexts (desktop, mobile, webmail, etc.).

2.4.2 Identity-based encryption: Identity-based encryption (IBE) [3], [10] aims to solve the problem of having to certify public keys for individuals, by instead offering the possibility of using a string representing their identity (e.g., their email address string) as the public key. An *identity provider* publishes a single public key (certified in the usual way), and then, for each registered email address, it computes a private key for the holder of the email address, and securely transmits it to him. The encryption primitive takes as input the provider's public key, the email address of the addressee, and the message, and returns a ciphertext. The ciphertext can now be decrypted using the private key given by the provider to the email address holder.

IBE is an attractive solution, because people are used to the idea that a person is represented by a human-readable string like an email address, rather than a public key. Unfortunately, in IBE the identity provider computes the private keys for all users, which means that the identity provider can decrypt any ciphertext: this is called the key-escrow problem. Key escrow can be considered reasonable in a corporate setting, where mail is owned by the organisation, but not in other settings. Another difficulty with IBE is key revocation, since the public key is the email address.

Certificateless encryption [19] solves the key-escrow problem of IBE by allowing users to create by themselves a public/private pair, which act in conjunction with, respectively, the public email address and private key created by the provider. In this setting, the encryption primitive takes as input the provider's public key, the addressee's additional public item, the email address of the addressee, and the message, and returns a ciphertext. The ciphertext can now be decrypted using the additional private item, and the private key given by the provider to the email address holder. The identity provider can't decrypt because it doesn't have the private item. The public item does not need to be certified, justifying the name "certificateless", because a third party that fakes the public item is not in possession of the private key from the identity provider and therefore cannot decrypt. A remaining weakness is that the identity provider can fake the public item, allowing it to mount "active" attacks, but this is still an improvement over IBE where the identity provider can passively decrypt. Certificateless encryption does not solve the revocation problem.

3 Certificate issuance and revocation transparency (CIRT)

We detail our extension of certificate transparency, in particular showing how a certificate authority can create efficient proofs that a given key is current (issued and not revoked).

3.1 Proving correct management of certificates

We propose a method which allows users of public keys to rely on certificate authorities without having to trust them. To put this another way, the method allows CAs to prove to users that they have behaved correctly. This solves the core problem related to certificate authorities. It also allows companies to provide end-to-end encrypted email in a form that is as user-friendly as ordinary email is today.

The method uses many ideas from certificate transparency (§2.2). In particular, a public append-only log is maintained of the certificates issued by a given certificate authority. In our method, the maintainer of the log can offer a proof that a certain certificate is *current* in the log, i.e., it has not been replaced or revoked. This is in contrast with certificate transparency, where proofs are that a certain certificate is present in the log, but not necessarily current. There are attempts to make certificate revocation work with certificate transparency, but as mentioned in §2.2.1 they require space/time which is linear (rather than logarithmic) in the number of certificates issued, and therefore the methods do not scale up. We describe and quantify this scalability aspect in §3.2.

A *certificate prover* (CP) is an entity that maintains a public log of certificates issued by a certificate authority. CP is able to issue proofs of extension of the log (that is, that the log is only ever appended), and proofs of currency of a given certificate. Suppose that CP's log consists of a collection of certificates:

$$db = [cert(Alice, pk_{Alice}), cert(Bob, pk_{Bob}), \ldots] .$$

To demonstrate its correct behaviour, CP must offer the services listed in Figure 3. It is important that these operations are done efficiently. More precisely, the data structure used for db must allow these operations to be done so that the time and transferred data is proportional to $O(\log n)$ or better, where n is the number of certificates stored.

The database of certificates is maintained as a pair of Merkle trees. In the first tree, items are stored left-to-right in chronological order, as in certificate transparency. We call this tree **ChronTree**. Certificates are added chronologically, by extending the tree to the right (see Figures 1 and 2). Revocation of a certificate is done by adding a new (perhaps null) key for the subject. Thus, a key for a subject is considered current

input	result
–	$h(db)$: the hash of the current database
$(subj, pk_{subj})$	Insertion: the certificate $cert(subj, pk_{subj})$ is inserted into the database.
$(subj, pk_{subj})$	Revocation: the certificate $cert(subj, pk_{subj})$ is marked as revoked in the database.
$h(db), h(db')$	Extension proof: a proof that db' is an append-only extension of db. We write this as $h(db) \sqsubseteq h(db')$
$h(db), subj$	Currency proof: a proof that $cert(subj, pk_{subj})$ is current according to db
$h(db), subj$	Absence proof: a proof that there are no certificates for $subj$ in db

Fig. 3. Services offered by a certificate prover.

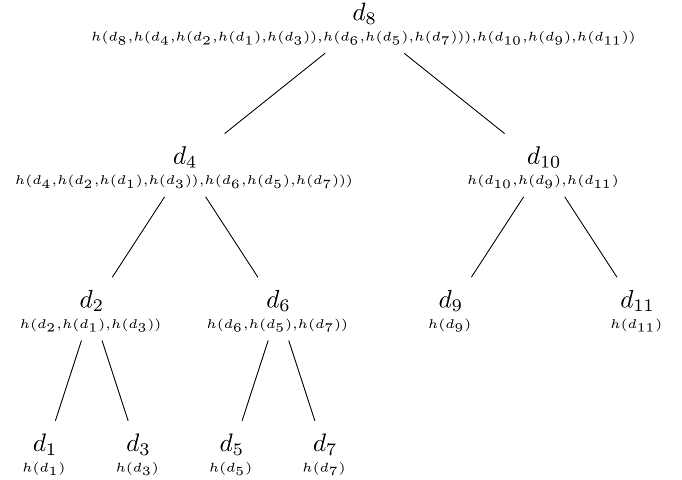

Fig. 4. An example of LexTree. The data items d_i have the form $\big(subj_i,\ (cert_{i,1}, cert_{i,2}, \ldots)\big)$, and $subj_1, \ldots, subj_{11}$ are in lexicographic order.

only if there is no later item for the subject. Using this tree, insertion, revocation and the extension proof are $O(\log n)$, by exploiting the properties of Merkle trees. However, as in certificate transparency, the currency proof is $O(n)$ because one has to show that a given key has not been revoked; this involves enumerating all the transactions that took place after the key was inserted. Similarly, an absence proof involves enumerating the whole tree.

To address this limitation, we additionally store the database as another Merkle tree, this time organised as a binary search tree, which we call **LexTree**. More precisely, the items $(subj, (pk_{subj,1}, pk_{subj,2}, \ldots))$ are stored at leaf and non-leaf nodes such that a left-right traversal yields the data in lexicographic order of the subject $subj$. Figure 4 shows an example, where the items d_i have the form

$$d_i = \big(subj_i,\ (cert_{i,1}, cert_{i,2}, \ldots)\big)$$

and $subj_1, \ldots, subj_{11}$ are in lexicographic order. The size of the list of certificates $(cert_{i,1}, cert_{i,2}, \ldots)$ is bounded by a constant N in LexTree; in other words, we only store up to $N-1$ revoked keys, and throw older ones away. A list of keys is stored for each subject, of which only the last one is the current one (the others are revoked). Here, insertion, revocation, currency proofs and absence proofs are $O(\log n)$. For example, a proof that $cert(Alice, pk_{Alice})$ is current consists of showing that $d = (Alice, L_{Alice})$ is in LexTree (which is done in logarithmic time, using standard Merkle tree proofs), and showing that pk_{Alice} is the last item in L_{Alice}, which is done in constant time since the length of L_{Alice} is bounded by a constant. Absence proofs (e.g., a proof that there are no keys for Bob in LexTree) can be done by showing that $(subj_1, L_{subj_1})$ and $(subj_2, L_{subj_2})$ are adjacent in the left-right traversal of LexTree, while lexicographically we have $subj_1 \leq Bob \leq subj_2$; this is also $O(\log n)$. However, to prove extension between db_1 and db_2 proof is now $O(n)$ because one has to consider each item that has been added between db_1 and db_2.

To obtain efficient proofs for both certificate currency and database extension, we use the two trees together. The database is a pair of Merkle trees **(ChronTree, LexTree)**. Insertion and revocation are done on both trees together, to ensure consistency. Extension and currency proofs are done using ChronTree and LexTree respectively, so all the operations may be done in time and space $O(\log n)$.

The definition of $h(db)$ is the Merkle hash value at the root of ChronTree, concatenated with that at the root of LexTree. One still has to verify that the two parts of the data structure are maintained consistently with each other. This verification requires $O(n)$ time and space, but it does not have to be computed by any particular user's browser. There are two ways that can be used to achieve this efficiently.

- **Random checking by users' client software.** The client software specifies a set of randomly chosen paths in ChronTree (resp. LexTree) and requests proof that the certificates on those paths are correctly present in LexTree (resp. ChronTree).
- **Public auditor.** The auditor receives all the updates from CP and maintains its own version of the two trees. It compares the $h(db)$ with the one reported by the log. Anyone can be a public auditor.

In summary, we extend certificate transparency by using two data structures, which are optimised for different kinds of proofs of transparency, and observers and users perform audits and random checks to ensure that the two data structures are maintained consistently. As in certificate transparency, linearity of the log is vital, and we use extension proofs and gossip protocols to ensure it (explained in §2.2).

Coverage of random checking: We briefly demonstrate that the random checking mentioned above is sufficient in terms of the likelihood of detecting cheating. Suppose we want to have a probability of 0.5 or more of achieving such detection.

Suppose there are n users, and each user logs in on average once per day, and one random check is made at each login. Then there are n random checks per day. Suppose a proportion v of the real users are "victims" (that is, out of n real users, the provider is cheating on nv of them by including a certificate for them in LexTree but not ChronTree, or ChronTree but not LexTree.) Then the probability of non-detection on a single check is $1 - v$, and the probability of non-detection within t days when there are n checks per day is

$$(1 - v)^{nt}$$

Suppose we set this at 0.5. Assuming that v is small (e.g., $0 \leq v \leq 0.1$), and approximating $\ln 2$ as 1, this is equivalent to:

$$nvt = 1$$

Thus, the time to detect cheating with probability 0.5 depends on n and v, and is better when both n and v are large. We plot two of these variables against each other (with the third one fixed, as indicated) in the graphs of Figure 5.

3.2 Space and time

In this section, we demonstrate the importance of the log proofs requiring space/time proportional to $O(\log n)$ rather than $O(n)$, by calculating some typical values. We suppose the database is required to store keys for one billion (10^9) subjects, who register with the service over a 10 year period. We also suppose that, on average, 5% of the keys are revoked each year. This amounts to 270,000 sign-ups per day and 140,000 revocations per day, a total of 410,000 transactions per day. Insertion and revocation each involve in the order of $\log_2 10^9 \approx 30$ operations on each tree. This will take negligible time.

Extension proof: Suppose a user has used the service and cached $h(db_1)$, and ten days later uses the service again and obtains $h(db_2)$. The user's software requests a proof that $h(db_1) \sqsubseteq h(db_2)$. This proof is provided by CP by comparing ChronTree_1 and ChronTree_2 corresponding to the two hashes. Thanks to the property of Merkle trees, the size of proof that CP provides is independent of the number of transactions that have taken place between db_1 and db_2 (in our example, it is about 1.4 million transactions). The proof consists of about 30 hash values, together with 30 other values. This is about 2 KB of data.

Currency proof: Suppose a user wishes to obtain the current key, with proof, for joeblogs@example.com. This proof is provided by CP using LexTree, which is also a Merkle tree. Because this tree is organised in order of subject identities, all the information about joeblogs is in the same place. CP merely has to prove the presence of the list of keys stored for joeblogs. Exploiting the properties of Merkle trees, the proof again consists of about 30 hashes and 30 other values, again 2 KB of data.

Necessity of both trees: Note that it is vital to store both trees. A currency proof done with LexTree, or an extension proof done with ChronTree, would be prohibitively expensive. To illustrate this, consider again the user that previously stored $h(db_1)$, and ten days later uses the service and obtains $h(db_2)$. The user's software requests a proof that $h(db_1) \sqsubseteq h(db_2)$, and the proof is provided by CP by comparing LexTree_1 and LexTree_2. Because the 4.1 million transactions that took place in the last 10 days are scattered throughout the tree, CP has to provide each transaction in turn along with the data required to verify it. This amount of data is 4.1 million times 2 KB, or about 10 GB. This is too much data and takes the user's software too long to download and process.

Consistency proof: Suppose an auditor wishes to check the consistency of the database represented by $h(db_2)$. The naive approach is to request a full account of all the sign-ups and revocations, and recompute (ChronTree, LexTree). This requires downloading all 10^9 certificates (which is in the order of $10^9 \times 60$ bytes, or 60 GB).

This can be improved considerably, but it is still $O(m)$ where m is the number of transactions that have taken place since the last audit. Suppose the auditor has previously conducted an audit for $h(db_1)$ done the previous day. The auditor now requests the transactions that have taken place in the last day, i.e., between $h(db_1)$ and $h(db_2)$. As mentioned, there are 410,000 transactions per day. He also requests the necessary parts of the Merkle trees to verify each transaction, one by one. As above, about 2 KB of data is required per transaction. So the auditor needs to download 800 MB per day. If he chooses to audit every hour instead, it is 30 MB of data for each audit.

Summary of method: Users efficiently verify short proofs that the certificate prover is honest in respect of the data of concern to the user (her own certificates and those of her associates). An auditor monitors larger proofs that the certificate prover is maintaining data structures consistently.

4 Application to email

The ideas of the preceding sections imply a way to manage email encryption key certificates which yields a system for end-to-end email encryption enjoying high degrees of security and user friendliness.

The core idea is that the email provider can at once be the certificate authority for its users, the maintainer of the CA log, and also the provider of storage for encrypted email. By using (our extension of) certificate transparency, the provider acting in this way is not required to be trusted by users.

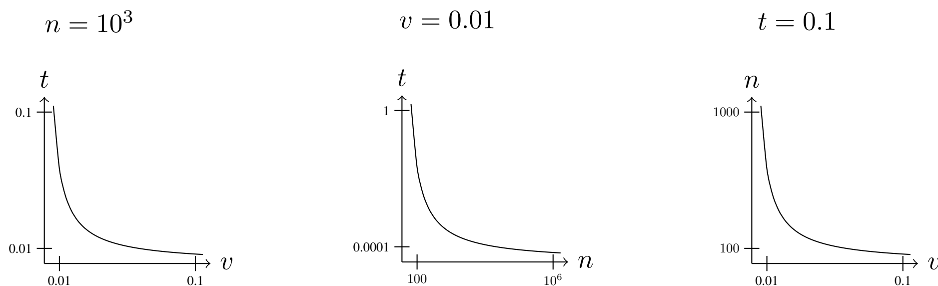

Fig. 5. Plots showing the coverage of random checking of the consistency of ChronTree and LexTree. Here, n is the number of users, and also the number of consistency checks per day. The time in days to detect inconsistency with probability 0.5 or greater is t, while v is the proportion of certificate subjects about whom the CA tries to cheat. The graphs show that cheating is detected within a few hours (0.1 days) provided there are enough subjects (e.g., more than 1000) and the victim rate v is not too small.

Although our provider is not required to be trusted, we assume it is not totally malicious either. Service providers today are large corporations, such as Google, Amazon, and Microsoft; they should not be trusted, although of course they do make great efforts to protect users' data from third-party attackers, and they have much to lose if those users take their custom elsewhere. The email system we detail below uses certificate transparency to ensure that the service provider cannot cheat without leaving evidence of its having cheated; moreover, that evidence is

- **Persistent:** the service provider cannot avoid leaving it, and cannot erase it;
- **Readily verifiable:** one can see that the evidence is indeed evidence that the service provider has cheated;
- **Transferable:** the evidence is meaningful to arbitrary observers, not just to the victim.

More precisely, the provider could, if it wished, certify a bogus key for a user, and then decrypt subsequent mail for the user. However, because this would be quickly detected, the provider will not launch such an attack. If the provider is a large organisation with a reputation to protect, it will not launch any attacks that could lead to evidence of its cheating.

4.1 The protocol

A user is assumed to have a mail provider (MP) that provides an email address and sending/receiving services (such as SMTP and IMAP), as well as email storage. The user also subscribes to a certificate prover service (CP). CP is a CA which maintains an certificate issuance and revocation transparency (CIRT) log of its certificates. In practice (as indicated above), we expect MP and CP to be the same provider. However, a user wishing to preserve an existing email address with an existing MP could use the services of a separate CP.

In brief, the protocol works as follows:

- Users have private/public keys, which are created and managed by the client email browser application.
- CP certifies the users' public keys, and maintains a database relating each public keys and email address[4]. It uses CIRT to maintain an append-only log of the certificates it issues and revokes.

Users' software automatically requests the log hashes and requests and validates proofs of extension and certificate currency, as detailed in the following sections.

4.1.1 Sign-up, sending, and receiving mail:

Alice signs up: Assume that Alice has downloaded an appropriate application, or installed an extension in her Outlook/Thunderbird, or is using an appropriately configured web app. For simplicity, we refer to Alice's client program as the application. At sign-up time, Alice's client software registers with CP her new or existing email address that she has with MP; then it creates her secret and public keys, and stores them in encrypted form with CP. The key for this encryption is noted k below. In more detail:

1) The application fetches current $h(db)$ from CP, and stores it.
2) Alice enters user-name, say "alice@example.com", and chooses a new password pw. The software chooses an encryption key k, which is stored securely on Alice's device. (Alternatively, to avoid storing k on the device, the authentication password pw and key k could be derived from a strong passphrase chosen by the user.)
3) CP creates an account for Alice, with user name "alice@example.com" and password pw.
4) The application creates public key pair pk_{Alice}, sk_{Alice}.
5) The application stores $(Alice, pk_{Alice}, \{h(db), sk_{Alice}, \dots\}_k)$ with CP.
6) The application makes a random check of log consistency, as detailed in §3.1.

In these steps, Alice's application stores her encrypted secret keys with CP, along with the current snapshot of the hash of the log which is also in the encrypted package. This is used later to verify that the log is correctly operated "append-only",

[4]Storing email addresses in the clear may be undesirable, for privacy and anti-spam reasons. To avoid this, the database and the logs and accompanying proofs can have hashes of addresses instead of real addresses.

and to prevent roll-back attacks in which CP sends Alice old versions of her cached information.

Alice sends email message to Bob:

1) Prior to authenticating Alice to CP, Alice's application fetches current $h(db')$ from CP.
2) The application retrieves its locally stored $h(db_s)$. Optionally, it requests proof that $h(db_s) \sqsubseteq h(db')$, and verifies the proof. (This verification is not necessary, since if it fails then a later verification will fail too; but if we do it now we detect any misbehaviour by CP slightly earlier.)
3) Alice requests and verifies proof that $cert(Alice, pk_{Alice})$ is current in db'.
4) The application authenticates Alice and fetches $(Alice, \{h(db), pk_{Alice}, sk_{Alice}, \ldots\}_k)$ from CP.
5) The application requests and verifies proof that $h(db_s) \sqsubseteq h(db)$ and $h(db) \sqsubseteq h(db')$. The application replaces its locally stored $h(db_s)$ with $h(db')$.
6) The application finds pk_{Bob} in db' and requests and verifies currency proof.
7) The application encrypts message for Bob with pk_{Bob} and sends it to him.
8) The application makes a random check of log consistency, as detailed in §3.1.

Step 1 and 2 ensure that CP is still maintaining the log in append-only fashion. In step 3, Alice's application verifies that CP is correctly maintaining her certificate. Step 5 ensures that the locally stored snapshot db_s is not later than the db stored in the user's account (db may in fact be later than db_s if the user has checked her email on a different device, and thereby updated db); and that the db stored in the account is prior to the current db'. These two checks prevent roll-back attacks, and attacks based on improper maintainance by CP of the log.

Bob receives mail from Alice: This process is similar. Bob's application retrieves his versions of $h(db_s)$, $h(db)$, and $h(db')$, and:

- checks $h(db_s) \sqsubseteq h(db) \sqsubseteq h(db')$.
- checks (Bob, pk_{Bob}) is correct in db'.
- gets pk_{Alice} from db', and requests currency proof.
- decrypts Alice's message and checks Alice's signature[5], if present.

4.2 Usability considerations

The system we describe here finally allows Johnny to encrypt his email (echoing the title of the classic 1999 paper mentioned earlier [23]). Just as a web user is in practice shielded from the requirement to have any real understanding of public keys and certificates, with these ideas an encrypted-email user can avoid having to understand the complexities of S/MIME and PGP.

[5]We didn't detail how the mail system supports digital signatures, but of course they're readily implemented too.

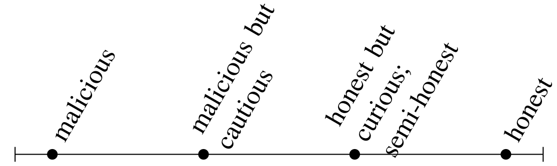

Fig. 7. A *malicious* provider is assumed to be willing to use any available strategy to attack, no matter what the consequence. A *malicious but cautious* provider is assumed to launch no attacks that would leave readily verifiable evidence of the attack. An *honest but curious* provider is assumed to launch no attacks that would leave any trace (whether verifiable or not); he confines himself to passive attacks. An *honest* provider is assumed to launch no attacks.

To use end-to-end encrypted mail that follows these ideas, users will download and install the application (or browser extension). As with any software, the user must download it from a trusted source. The user launches the software, and configure it to access their existing mail account (or set up a new account). This step is the same as configuring any mail software.

When the user starts her email browser, it optionally shows one or more icons (or perhaps "lights"; see Figure 6) indicating the result of a consistency check of the service provider. Each icon corresponds to the result of a check made by an auditor that the user has subscribed to. The icon will display a visual representation (for example, the light is coloured green or red) indicating "healthy" or "problem". The user can sign up to whatever auditor he likes, by appropriately configuring the browser. The user can be his own auditor.

To send a message, the user enters the email address as usual, assisted by a contacts manager and autocompletion in the usual way. It's vital to be sure to send the mail to the intended address, since the address determines the encryption key that the application will select (and verify the proofs about). This is the counterpart in PGP of being sure to have the right public key in her keychain with the right trust level in its signers, except here it is something the user can understand. It is natural to users that if they send a message to the wrong recipient, then confidentiality of the message may be lost.

The application handles recipients for whom there are no public keys (in this case the log produces a proof of absence of any certificate for that user[6]). The application displays by means of a visual indicator (e.g., by colouring the address) whether the message to that recipient will be encrypted or not.

In the envisioned GUI, there is no encrypt button and no decrypt button. Messages are encrypted or decrypted automatically in the cases in which the CIRT infrastructure reports an appropriate key. There are no user dialogues or messages that refer to keys or certificates.

[6]Note that a downgrading attack is impossible: absence is not failure to prove presence, but is a proof of absence.

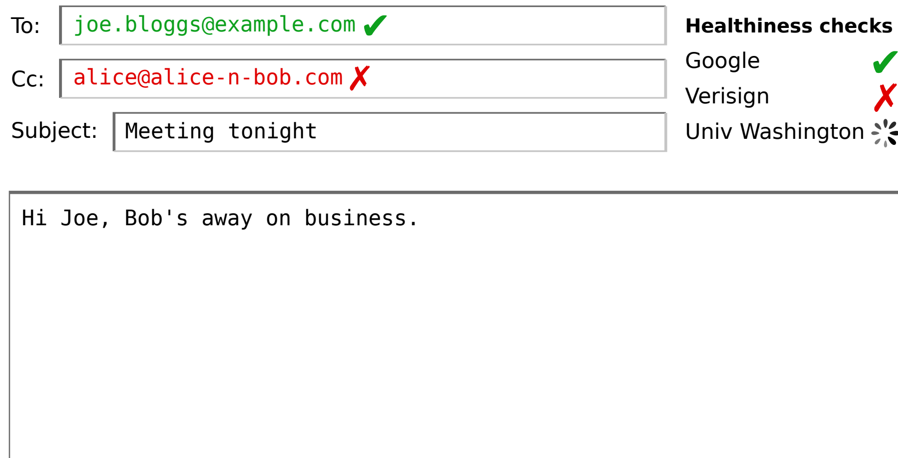

Fig. 6. Email user interface. Visual symbols and/or colours show whether the email will be encrypted for the recipient. On the right, some auditor reports are available showing the correctness status of CP's log.

4.3 Key and password management

As mentioned in the signing-up section of §4.1.1, there are two options for arranging user authentication.

- Users' passwords are high-entropy. In this option, the user's password pw is a high-entropy password, and not disclosed to the server. The user authenticate to server with $kdf(pw, 1)$ for a suitable key derivation function kdf, and uses $k = kdf(pw, 2)$ as the key purse encryption key. The password has to be high-entropy to prevent the server (or anyone else) performing guessing attacks to obtain k.
- Users' passwords are known to the server. In this option, the key purse encryption key k is stored on the user's device. The password need not be high-entropy because the server can prevent on-line guessing attacks.

These two alternatives are fundemantal to any cloud computing application in which users have encryption keys which are confidential from the cloud provider; Wuala [27] and ConfiChair [31] are examples, and their designers have the same two options.

4.3.1 High entropy password: This option is the most flexible, since the user can access the services from any device without needing to provision it with the key purse encryption key k. The main disadvantage, however, is that the server can try offline guessing attack on pw in order to derive k.

If users want to change their password, this can be done easily: the client application need only decrypt and re-encrypt key purse, using the keys derived from the old and new passwords respectively.

If a user loses her password, the system can't offer any recovery mechanism (as in Wuala [27]). At best, the user can prove ownership of the account by out-of-band means; this will allow her to revoke her public key and re-initialise the account, but she won't have access to her existing email store.

4.3.2 Device key: This option is more secure, but it requires a means to migrate k to new devices. We detail such a protocol in Figure 8, based on SPEKE [37].

Requests from a user to change her password are handled by the usual means; requests to change the k are handled by decrypting the key purse with the old k, and encrypting with the new one. In the case of a lost password, the usual kind of recovery mechanisms can be used.

If the key k is lost, then the user loses access to their historic data, but can use knowledge of their password to prove ownership of the account; as above, this will allow her to revoke her public key and re-initialise the account. Note, however, that since the user will typically have k on multiple devices, it is unlikely that she loses it completely.

5 Conclusions

5.1 Summary

We have extended certificate transparency to handle revocation efficiently, resulting in a system we call *certificate issuance and revocation transparency*. This contributes to its usefulness on the web. We apply this certificated transparency to email, allowing an email provider to certify keys for its users without requiring them to trust it. This yields a system for end-to-end encrypted email which is both usable (users don't have to understand anything about keys or certificates, or take special actions), and secure (mail is end-to-end encrypted and there are no third parties required to be trusted). In contrast with S/MIME, PGP, IBE and certificateless encryption, the CA (or identity provider) is prevented from launching attacks on its users. This means that end-to-end email encryption can finally be made as user-friendly and accessible as secure web browsing.

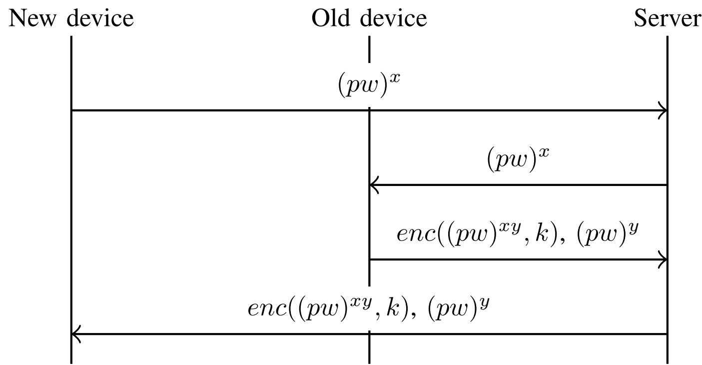

Fig. 8. Protocol to allow migration of the encryption key k from one device to another, based on SPEKE [37]. The user creates a request on her new device $(pw)^x$ based on a randomly chosen x; here we assume pw is a representation of the password in a suitable Schnorr group. Next, the user's application on her old device retrieves the request from the server, and creates an encryption key $((pw)^x)^y$ by selecting another random y. The key k is encrypted with $(pw)^{xy}$ and sent to the server along with $(pw)^y$. Then the user returns to the new device to retrieve that information, decrypt k, and install it on the device.

5.2 Discussion: cloud

Underlying these ideas is an attacker model appropriate for cloud computing. In most cloud-based applications today, users are required to fully trust the cloud provider. "Fully trusted" is unacceptably optimistic; researchers are attempting to change that, for example with fully homomorphic encryption, so that, on the contrary, the cloud provider could be considered fully malicious. But that is unduly pessimistic. Cloud providers are large organisations with reputations to preserve, and they compete to attract customers. Therefore, they will not attack their users at any cost; they will not launch attacks that leave verifiable evidence. Thus, they are in reality somewhere between the extremes of "malicious" and "trustworthy" (Figure 7). "Honest-but-curious" already lies between these extremes; it says that the attacker launches passive attacks but not active ones. However, there is no reason to suppose a cloud provider will refrain from active attacks. We adopt the term "malicious-but-cautious"; the cloud provider is assumed to be malicious if he can get away with it, but cautious in not leaving any verifiable evidence of its misbehaviour. This attacker model is related to the *covert adversary* of [22], but it additionally assumes that the cloud provider acts to protect its users from third-party attacks.

Systems based on this model (such as the email system we detailed) deny the possibility of monetising users' data, e.g. for content-related advertising, as pioneered by Google and now done by other providers. One might ask whether providers would be willing to go on providing hosting services for free, without this revenue opportunity. That question is beyond the scope of the paper, but nevertheless we can't resist speculating about it. The most successful internet companies today offered services long before they had any idea how they could be monetised, so we don't expect that to be an obstacle in practice. Moreover, user applications may be willing to leak some data to the provider, such as the fact that a particular message mentions "hotel" and "Paris", allowing the provider to serve adverts for hotels in Paris without having the whole plaintext message. Finally, we expect that when users fully realise the consequences of paying for services with their data, they will prefer to pay modest amounts of money and keep their data private.

A more serious obstacle to take-up of such an email system may appear to be spam. If mail is decrypted by the receiver, it prevents the server from deleting messages after applying spam detection. This requires spam handling on the client side, which is less convenient than handling it on the server. Spam can also be mitigated if users configure their mail browsers not to accept encrypted mail unless it is also signed by users they are already in contact with.

5.3 Future work

In future work, we intend to perform rigorous security analysis of certificate issuance and transparency, in the malicious-but-cautious model. That involves defining the model formally. We will also analyse the email protocol in that model.

References

[1] A. Langley. Public-key pinning. *ImperialViolet* (blog), May 2011.

[2] A. Langley, "Revocation checking and Chrome's CRL," *Imperial Violet* (blog), Feb 2012.

[3] A. Shamir. Identity-based cryptosystems and signature schemes. CRYPTO 1984.

[4] B. Amann, M. Vallentin, S. Hall, and R. Sommer. Revisiting SSL: A large-scale study of the internet's most trusted protocol. Technical report, ICSI, 2012.

[5] B. Laurie and E. Kasper. Revocation transparency. Google Research, September 2012. www.links.org/files/RevocationTransparency.pdf.

[6] B. Laurie, E. Kasper, and A. Langley. Internet-draft: Certificate transparency. 9, March 2013.

[7] "Black Tulip Report of the investigation into the DigiNotar Certificate Authority breach," Fox-IT (Tech. Report), 13 Aug 2012.

[8] C. Arthur, "Rogue web certificate could have been used to attack Iran dissidents," The Guardian, 30 Aug 2011.

[9] Certificate transparency. www.certificate-transparency.org.

[10] D. Boneh and M. K. Franklin, Identity-based encryption from the weil pairing. CRYPTO 2001.

[11] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In USENIX Annual Tech, 2008.

[12] Electronic Frontier Foundation. Iranian hackers obtain fraudulent HTTPS certificates: How close to a Web security meltdown did we get? https://www.eff.org/deeplinks/2011/03/iranian-hackers-obtain-fraudulent-https .

[13] J. Clark and P. C. van Oorschot. SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. IEEE Symposium on Security and Privacy, 2013.

[14] M. Alicherry and A. D. Keromytis. Doublecheck: Multipath verification against man-in-the-middle attacks. In ISCC, 2009.

[15] M. Jelasity, S. Voulgaris, R. Guerraoui, A. M. Kermarrec, and M. Van Steen. Gossip-based peer sampling. ACM Transactions on Computer Systems (TOCS) 25(3), 2007.

[16] MS01-017: Erroneous VeriSign-Issued Digital Certificates Pose Spoofing Hazard http://support.microsoft.com/kb/293818 .

[17] P. Eckersley and J. Burns. Is the SSLiverse a safe place? In Chaos Communication Congress, 2010.

[18] R. Rivest. Can we eliminate certificate revocation lists? In *Financial Cryptography*, 1998.

[19] S. S. Al-Riyami and K. G. Paterson. Certificateless public key cryptography, ASIACRYPT 2003.

[20] The Register. Trustwave admits crafting SSL snooping certificate; Allowing bosses to spy on staff was wrong, says security biz. www.theregister.co.uk/2012/02/09/tustwave_disavows_mitm_digital_cert .

[21] Trust assertions for certificate keys (TACK). Internet draft, 2012.

[22] Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. Theory of Cryptography, 2007.

[23] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: a usability evaluation of PGP 5.0. Proceedings of the 8th conference on USENIX Security Symposium, 1999.

[24] Certificate patrol. http://patrol.psyced.org.

[25] The EFF SSL Observatory. https://www.eff.org/observatory.

[26] J. Appelbaum, "Detecting Certificate Authority compromises and web browser collusion," Tor Blog, 22 Mar 2011.

[27] Wuala, an encrypted cloud-based store in which users' encryption keys are not disclosed to the cloud provider. http://www.wuala.com.

[28] N. J. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. Schuldt. On the security of rc4 in tls and wpa. 2013.

[29] N. J. AlFardan and K. G. Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. *IEEE Symposium on Security and Privacy*, 2013.

[30] M. Alicherry and A. D. Keromytis. Doublecheck: Multipath verification against man-in-the-middle attacks. In *ISCC*, pages 557–563, 2009.

[31] M. Arapinis, S. Bursuc, and M. Ryan. Privacy supporting cloud computing: Confichair, a case study. In *Principles of Security and Trust*, pages 89–108. Springer, 2012.

[32] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008. Updated by RFC 6818.

[33] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[34] P. Eckersley. Internet-draft: Sovereign key cryptography for internet domains. 2012.

[35] N. Falliere, L. O. Murchu, , and E. Chien. W32.stuxnet dossier. technical report, symantec corporation, 2011.

[36] P. Hoffman and J. Schlyter. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698 (Proposed Standard), Aug. 2012.

[37] D. P. Jablon. Extended password key exchange protocols immune to dictionary attack. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 1997., Proceedings Sixth IEEE workshops on*, pages 248–255. IEEE, 1997.

[38] J. Kasten, E. Wustrow, and J. A. Halderman. CAge: Taming certicate authorities by inferring restricted scopes. In *Financial Cryptography*, 2013.

[39] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor. Accountable key infrastructure (AKI): A proposal for a public-key validation infrastructure. In *the 22nd International World Wide Web Conference (WWW 2013)*, 2013.

[40] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962 (Experimental), 2013.

[41] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter. Public keys. In *CRYPTO'12*, pages 626–642, 2012.

[42] M. Marlinspike. SSL and the future of authenticity. In *Black Hat, USA*, 2011.

[43] M. Marlinspike and T. Perrin. Internet-draft: Trust assertions for certificate keys (TACK). 2012.

[44] P. Roberts. Phony SSL certificates issued for Google, Yahoo, Skype, others. Threat Post, March 2011.

[45] C. Soghoian and S. Stamm. Certified lies: Detecting and defeating government interception attacks against SSL. In *Financial Cryptography*, pages 250–259, 2011.

[46] T. Sterling. Second firm warns of concern after dutch hack. Yahoo! News, September 2011.

[47] S. Turner and T. Polk. Prohibiting secure sockets layer (SSL) version 2.0. RFC 6176 (Proposed Standard), Mar. 2011.

[48] D. Wendlandt, D. G. Andersen, and A. Perrig. *Perspectives:* improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference*, pages 321–334, 2008.