PowMail: Want to fork? Do some work.

by

Alin Tomescu

B.S., Stony Brook University (2012)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

Signature redacted

Author .....
Department of Electrical Engineering and Computer Science
August 28, 2015

Signature redacted

Certified by .....
Srinivas Devadas
Professor
Thesis Supervisor

Signature redacted

Accepted by .....
Leslie A. Kolodziejski
Professor
Chair, Department Committee on Graduate Theses

PowMail: Want to fork? Do some work.

by

Alin Tomescu

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 2015, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

We design and implement PowMail, a secure email framework that uses proof-of-work to prevent impersonation attacks. PowMail uses a novel authenticated data structure called a *PoW-tree* to enable email users to jointly certify the history of all users' public keys. An implementation is given in the Go programming language. We evaluate PowMail and demonstrate its feasibility for securing email communications today.

Thesis Supervisor: Srinivas Devadas
Title: Professor

Acknowledgments

To my dear friend Ilusha; my muse, my rock, my inspiration. This thesis would have been impossible without your help.

First of all, I would like to thank my advisor, Professor Srinivas Devadas, for guiding me towards this work, for his constant support, wisdom and inspiration. I would also like to thank Mashael Saad Al-Sabah for giving me the opportunity to help with her secure email project and for our many interesting discussions that led to this work. I would like to acknowledge the constant support of my family, my friends and my girlfriend. Without their love and kindness, I wouldn't have had the strength. Finally, I would like to thank our Computation Structures Group at MIT CSAIL for all their useful feedback and productive discussions that steered this work into a unique direction.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, we address the problem of end-to-end email security in the presence of powerful adversaries. While exiting email privacy techniques exist and are used today (PGP[55], S/MIME[11], STARTTLS[43]), we argue that they are not sufficient for security in the face of strong adversaries who can compromise email service providers or trusted entities such as certificate authorities. We propose a proof-of-work-based solution for end-to-end email security. We believe our approach could be effective at preventing impersonation attacks early by requiring attackers to expend a lot of computational power. We believe that making attacks computationally and thus monetarily expensive is a viable way of securing the email network. Our approach does not share the disadvantages of blockchain-based solutions such as high-bandwidth and storage requirements, nor does it require users to engage in competitive mining[1].

Roadmap. In this chapter, we describe our motivation, give some relevant background and present the challenges of building a secure email system. In chapter 2, we describe the actors in our system, our threat model assumptions and our goals. In chapter 3, we present existing work and explain why it is unsuitable for email security. In chapter 4, we detail our system's architecture and in chapter 5, we analyze the security of our system. In chapter 6, we describe our implementation and we evaluate it in chapter 7. We address future work in chapter 8 and conclude in chapter 9.

[1]Users still have to mine, but they can profit without being the first to find the proof-of-work.

1.1 Motivation

It is well known that users share sensitive data over email[33] and recent email leaks serve as a constant reminder of that fact[53, 30, 32]. Thus, we believe email security is important and needs to be addressed.

Firstly, encrypting and authenticating email in transit is important to preserve secrecy and integrity of messages transmitted across an insecure network. Emails can easily be snooped on or tampered with as they are transmitted. Encrypting emails stored on service providers is crucial for preventing data leaks when service providers are compromised[9, 45]. Secondly, authenticating the senders of emails can prevent impersonation attacks and phishing attacks[14]. Thirdly, signing emails using non-repudiable digital signatures can enable users to close contracts over email or enter legal agreements over email. Note that due to concerns over digitally signed emails[17, 20, 14], non-repudiable signatures should be strictly optional, while message authentication can be performed by default using deniable message authentication codes (MACs).

1.2 Background

1.2.1 Our friends: Alice and Bob

In this work, we will often discuss two fictional characters Alice and Bob who want to communicate securely via email. Alice and Bob are two email users with email accounts `alice@wonderland.com` and `bob@puzzleworld.com` respectively. In addition, we will sometimes use Mallory to refer to active network adversaries. Mallory can snoop on network messages, tamper with messages, and he can create, duplicate or redirect messages on the network.

1.2.2 Public key cryptography

In public key cryptography, Alice can simply obtain Bob's *public key* (PK) and encrypt[2] a message for Bob. Bob is the only one who can decrypt the message using his *secret key* (SK) which only he knows. If the message were an email message, then Alice's email client can *transparently* look up Bob's public key without imposing additional work onto Alice. User transparency is an important goal (see §2.6), as it enables fast adoption of a secure system. We believe using public key cryptography is fitting for email communication, where there is a pre-established expectation that users be able to send an email to anyone in the world simply by obtaining the recipient's email address.

1.2.3 Impersonation

Impersonation is a central problem in public key cryptography. While users can easily generate a public key and its corresponding secret key (i.e. a key-pair), it remains difficult to securely obtain *someone else's* public key. Simply put, since anyone can generate a key-pair, Alice cannot know whether the public key PK_B she obtained is indeed Bob's public key. For example, Mallory can generate a key-pair $(PK_{B'}, SK_{B'})$ and impersonate Bob by sending the generated public key $PK_{B'}$ to Alice. Alice will think $PK_{B'}$ is Bob's public key and use it to encrypt messages for Bob. However, all her encrypted messages will be decrypted by Mallory who has $SK_{B'}$. Furthermore, instead of dropping the encrypted message, Mallory can reencrypt the message for Bob under the real PK_B and forward it to him, thereby avoiding detection by Bob, who could otherwise get suspicious about not receiving messages from Alice.

Importantly, note that impersonation is a fundamental problem for public key cryptography: *the binding between a user and their public key is not guaranteed to be correct.* In order to enforce a correct, verifiable binding, additional techniques need to be employed. For instance, in TLS[52], the binding between a domain such as `google.com` and its public key is enforced by certificate authorities (CAs) via a

[2]In this thesis, we always mean *encrypt and authenticate* when talking about encryption

cryptographic proof of the binding known as a *certificate*. A certificate consists of a signature under the secret key of a CA on the domain's name and its public key. A certificate is only issued by a CA after it has verified the true identity of the requesting domain[3]. The certificate is verifiable by anyone in possession of the CA's public key. The certificate is an unforgeable statement by the CA, since only the CA knows its secret key used for signing certificates. The certificate attests that a specific domain is the owner of a specific public key, enabling users to trust this binding is correct as long as they trust the CA and have the CA's public key. In TLS, there are hundreds of CAs who can sign certificates. Users will verify certificates from any CA and, if the signature is correct, they will trust that the CA is not lying about the binding.

Unfortunately, certificate authorities do not fully solve the impersonation problem; they only make it smaller. CAs act as gatekeepers, preventing fake bindings from being created by verifying the domains who request certificates. However, certificate authorities may still *equivocate* (or lie) about public key bindings; all they have to do is sign another certificate. Of course, CAs are trusted not to do so, but we argue that such trust is misplaced.

1.3 Challenges

Below, we describe the main challenges in building a secure email system. We argue that equivocation remains a challenge simply because CAs are not trustworthy and that finding an appropriate trust model, while ensuring scalability of our system, is difficult.

1.3.1 Equivocation

As discussed in §1.2.3, TLS certificate authorities can impersonate users by equivocating, enabling man-in-the-middle attacks and resulting in loss of user privacy. Even if CAs were trusted not to equivocate, they may still be hacked[16, 37] and, to make

[3]This verification can be performed in-person for high-assurance certificates.

things worse, there is circumstantial evidence to suggest that CAs are susceptible to government coercion[22, 48, 44].

Unfortunately, TLS does not provide a way for websites to check what certificates have been issued in their name, making detection of equivocation difficult. While existing techniques like Convergence[2] and Perspectives[5] can sometimes detect equivocation, they are far from being foolproof and require user intervention and knowledge. One promising solution is Certificate Transparency (CT)[36], which was deployed by Google and is being used by modern day browsers such as Google Chrome. We will discuss CT and other similar approaches in §3.5. We believe that preventing equivocation is the main challenge in building a secure email system.

1.3.2 Trust

One of the problems with the CA system in TLS is the lack of trust agility[12]: users cannot decide that certain CAs are untrustworthy. Instead, users are required to trust all CAs in TLS not to engage in impersonation attacks. This leads to a weakest-link security model for TLS, where if one CA is compromised, then the entire CA system is compromised[51]. We believe it is important to avoid placing too much trust in our system's actors. Thus, we seek to *audit* the actors in our system and verify that they behaved correctly. Note that, ultimately, trust in the email users to check that they are not being impersonated is still required.

1.3.3 Scalability

The size of the email network and the number of emails being sent every day mandate a scalable solution for the email security problem. The simple requirement that email users need to be able to look up each other's public keys creates a high bandwidth requirement and a storage requirement. Our solution needs to address this.

1.3.4 User interface

We recognize the paramount importance of having a good user interface to a secure email system. However, this thesis does not address this problem. We choose to focus on how to prevent equivocation in a secure email system and defer the user interface problem to future work.

1.4 Summary

In this chapter, we described why email security is an important problem. We provided some background on public key cryptography and explained why it is necessary for email security. We discussed unique challenges that arise in email security. Most importantly, we introduced the *equivocation problem* as a central problem in email security.

Chapter 2

The secure email problem

2.1 Actors

Email communication involves several actors. First, *email users* want to communicate securely and quickly by sending each other electronic mail. Second, *service providers* (SPs) enable email users to communicate. An email service provider allows users to register for an email account. Email service providers send and receive emails on behalf of their users. SPs also provide persistent storage for user emails, giving each user convenient access to all her past emails. Finally, SPs act as initial *identity providers* for their users. An SP can help its users authenticate themselves to a certificate authority for the purpose of obtaining a certificate. Secure email communication requires an additional actor called a *key server* which replaces trusted certificate authorities. Key servers are very similar to certificate authorities, except that they are *auditable*, enabling email users to check if they have been impersonated.

2.2 Key servers

Key servers allow users to register public key bindings for themselves and to efficiently and securely look up public keys. Instead of signing individual bindings such as (Alice, PK_{Alice}), a key server commits to the *history* of all bindings it has signed. The purpose of committing to the entire history is to allow any user to discover if they

have been impersonated. For instance, Alice would quickly find out if she has been impersonated by discovering a fake binding in the history committed by the key server. Most importantly, key servers are *auditable*: they enable users to monitor their own bindings to detect if they are being impersonated.

2.3 Threat model

Email Service Providers. Email service providers are not trusted in our system. They may attempt to read the emails of their users and to impersonate their users but they will be detected. They can be compromised by attackers or by corrupt employees.

Key Servers. Key servers can be compromised by attackers. Key servers are not trusted to respond correctly to public key lookups. Instead, users will verify proofs generated by a key server to ensure that the key server is behaving correctly. Key servers can try to equivocate about users' bindings as described in §1.2.3.

Network adversaries. We assume the existence of both passive and active network adversaries. Network adversaries can snoop on messages, redirect messages, duplicate messages or create new messages.

Powerful adversaries. We assume the existence of powerful adversaries who can coerce system actors into misbehaving or can compromise actors. Our goal is to prevent equivocation in the face of such adversaries.

Email users. Email users are trusted to verify that key servers respond correctly to public key lookups and that key servers are not impersonating them. We assume users access their email accounts from uncompromised machines, which is a requirement for end-to-end secure communication where compromise of an endpoint would immediately lead to compromise of the secret keys used to secure the communication.

2.4 Security goals for key servers.

Key servers keep track of the users' public key history and cryptographically commit to this history. We describe our security requirements for key servers below.

Commitment efficiency: The server's commitment of the history is small enough for Alice to download and store.

Query efficiency: Alice can search the history quickly [1].

Query correctness: Alice can verify her search results against the history via a (non)membership proof.

Append-only: When the server commits to a newer history, the previous history needs to be fully included in the new one. The server should not be able to remove or change PKs in the committed history and then commit to a new history, which would be *inconsistent* with the previous one.

Secure binding update: After Alice has registered a public key for herself, thereby creating a binding, only Alice should be able to update her binding.

Freshness: The server cannot withhold the latest history and prevent Alice from finding out about fake bindings.

Non-equivocation: The server cannot easily *fork* the history and show Alice a *view* where her fake PK is not present, while showing Bob a view where her fake PK is present.

Fork-consistency: If the server forks the history, then the server can never unfork it later [39].

Proof of misbehaviour: If the server misbehaves by creating a fake binding, the misbehaviour is *evident* (i.e. users can quickly find out about it) and provable.

2.5 Security goals for email transmission.

Authentication, integrity and secrecy. Email messages should be sent and stored with secrecy and integrity. Email senders as well as recipients should be authenticated. If an email is claimed to have been sent by Alice, then there must be cryptographic proof attesting this. Similarly, if Alice is sending an email to Bob, then she needs to be sure only Bob can decrypt that email.

[1] We aim towards $O(\log n)$-time when there are n bindings in the history.

Forward-secrecy. Compromise of user's long-term secret keys should not affect the confidentiality of previously exchanged emails[20]. Each email[2] should be encrypted with a symmetric key that is generated independently of the communicating users' key-pairs.

Deniable and non-deniable authentication. Previous work has expressed concern over digitally signing emails[17, 20, 14]. Thus, our goal is to allow users to authenticate each other either with deniability or non-deniability. Non-deniability means Bob can prove that Alice sent him a message and Alice cannot deny it and can be achieved by signing emails before sending them. Deniability means Bob cannot prove Alice sent him a message and can be achieved if Alice only MACs her message to Bob, preventing Bob from proving that Alice sent him that message, since Bob could have computed the MAC as well.

Note that currently deployed systems like DKIM[26] can break non-deniability, as DKIM digitally signs all outgoing message of a service providers. In that case, even if Alice only MACs her messages, her service provider signs all outgoing email, including hers. This seems to take away from Alice's deniability to some extent. Even though her MAC is deniable, the service provider's signature suggests that Alice did send the message[3].

2.6 Functional goals

User-transparency. Security should not impose additional work for users. On the contrary, users should experience security as a transparent process, similar to TLS. This increases adoption of the system and prevents users from misusing the system and breaking their security.

Deployability. Our system must be deployable in order to achieve its security goal. For example, if Alice is using our system and wants to send a secure email to Bob, but Bob's email provider has not adopted our system, then Alice will have to send

[2]With the exception of the first sent email (see §4.8).

[3]Assuming the SP is not misbehaving, of course.

Bob the email in plaintext thereby losing desirable properties such as secrecy and authentication.

Performance. Users are accustomed to sending emails quickly and receiving emails instantly after they are sent. Our goal is to maintain the same speed of email delivery while securing it.

Deployment flexibility. Users should not have to wait for their providers to deploy our system before they are able to use it. Either no deployment should be necessary from service providers or incremental deployment should be possible. We adopt this goal from Lightweight Email Signatures (LES)[14].

Scalability. There are nearly 100 billion legitimate emails being sent everyday[31]. Our system should be scalable in order to handle the load of the email network. Public key lookup should be efficient as many users will be monitoring their own public keys and looking up other users' public keys. Similarly, public key registration and update should be fast, since many users will want to register public keys or updated existing ones.

2.7 Summary

In this chapter, we introduced the main actors that interact in a secure email system. We introduced a new actor called a *key server* that behaves similarly to a certificate authority but can be audited to detect impersonation attacks. We postulated a set of goals that a secure email system should achieve and detailed the threat model under which the system should remain secure.

Chapter 3

Existing work

3.1 Secure/Multipurpose Internet Mail Extensions (S/MIME)

S/MIME[11] relies on certificate authorities (CAs) to vouch for the binding between a user and her public key. In S/MIME, Alice is required to trust all TLS certificate authorities not to impersonate her. If Alice is willing to trust all CAs, she can request an email certificate from one CA and give that certificate to Bob. Likewise, if Bob trusts the CA not to impersonate Alice, then he can use the certificate to extract Alice's public key and send her a secure message.

S/MIME does not specify a mechanism for users to look up each other's public keys and their certificates. Therefore, users have to resort to publishing their S/MIME certificates on a public forum such as their personal websites. This inhibits adoption of S/MIME due to the lack of user transparency (see §2.6). Another problem with S/MIME is that it requires too much trust in CAs. If a CA misbehaves, there is no easy way to make this evident to users. A CA may simply sign a fake certificate for Alice, give it to other users and decrypt all incoming messages for Alice.

To conclude, since S/MIME does not consider how users are to obtain each other's public keys and does not offer a mechanism to prevent equivocation, we cannot consider it a secure email solution.

3.2 Pretty Good Privacy (PGP)

PGP is a popular email security tool developed in 1991 by Phil Zimmermann[55]. PGP allows users to encrypt and sign their emails. PGP tries to solve the impersonation problem without certificate authorities via a decentralized Web of Trust (WoT). The WoT relies on social networks instead of certificate authorities to attest the binding between users and their public keys. For instance, if Alice wants to certify her binding, she needs to obtain signatures from other users she knows and trusts. Alice can then upload her binding along with its signatures to a key server such as MIT's PGP key server[3]. Bob can look up Alice's key on MIT's key server and download it along with its signatures from other people. For Bob to validate Alice's binding, Bob needs to trust the signers of Alice's binding, or find a *certification path* from users he trusts to the users who signed Alice's certificate. However, such certification paths dilute trust as they may contain untrustworthy users. For instance, if Bob trusts Charlie, and Charlie trusts Dan, it is not clear that Bob should trust Dan as well: trust is not always transitive. PGP leaves the decision of trusting Dan to Bob but this can be confusing for Bob who might be an inexperienced user lacking an understanding of PGP's trust model. We argue that the WoT model makes PGP hard to use, as it is difficult to quantify trust in social structures[54].

3.3 Lightweight Email Signatures (LES)

Lightweight Email Signatures (LES)[14] is an extension to DomainKeys Identified Mail (DKIM) whose main goal is to prevent against email spoofing by digitally signing all outgoing emails. For instance if Alice has a Gmail account and sends an email to Bob, then Gmail, a DKIM-enabled provider, will sign her email to Bob. Bob can verify the signature by obtaining Gmail's public key from the Domain Name System (DNS) where each service provider will store their public key in a DNS TXT record. After verifying the signature, Bob will know the email indeed came from a Gmail user. If Bob trusts Gmail not to lie about the email's sender, then Bob is assured that the

email did indeed come from Alice. Note that DKIM is deployed and used today and can be effective at preventing phishing attacks.

LES is similar to DKIM with the exception that it relies on identity-based cryptography (IBC)[49] in place of normal public key cryptography. By using IBC, LES can be deployed in settings where DKIM would not work because the email service provider cannot always differentiate between its users. For instance, MIT's SMTP server does not authenticate email senders. As a result, MIT cannot use DKIM to sign outgoing emails since the SMTP server would not know who the real sender of the signed email is.

LES uses identity based cryptography to distribute key-pairs to email users and solve the SMTP authentication problem explained above. An email provider has a master public key (MPK) and a master secret key (MSK). The provider can use its MSK to generate a key-pair for any user. A user can obtain any other user's public key if she has their email address and their provider's MPK. The MPK of providers are stored in the DNS system and can be obtained easily. As a result, once users have obtained their key-pairs, they can send each other digitally signed emails and authenticate each other.

LES also addresses privacy concerns regarding the use of non-repudiable signatures in DKIM. In LES, users are allowed to obtain *evaporating key-pairs* whose corresponding secret key is revealed after a certain time interval. Evaporating keys turn a non-repudiable digital signature into a repudiable one, because revealing the secret key implies other users could have used it to sign a message.

Much like DKIM, LES-enabled email providers are trusted not to impersonate their users, since they can derive any user's SK using the MSK. Unfortunately, there is no way to verify that providers are not engaging in impersonation attacks.

3.4 Lightweight Email Encryption (LWE)

Lightweight Email Encryption (LWE)[15] is an email encryption solution similar to LES[14] (see §3.3). LWE uses identity-based cryptography and the DNS system to

distribute public keys, but adjusts the threat model in LES to prevent providers from learning their users' secret keys. Email providers should not be able to learn their users' secret keys, otherwise they could easily decrypt their users' emails. Also, compromise of an email service provider should not lead to an attacker learning the master secret key (MSK). This would be disastrous, as an attacker who obtains the provider's MSK can decrypt all communication of all users.

To prevent service providers from learning user secret keys, LWE allows users to add their own shares to their secret keys. Note that this will also change the public keys corresponding to the secret key, which means the provider's MPK is not sufficient to derive a user's PK; users will also need the additional share added to the key, meaning users must publish their shares and disseminate them to other users. To prevent service provider compromises from revealing the provider's MSK, LWE splits the MSK on multiple machines. As a result, compromising a single machine will not reveal the MSK.

LWE does not defend against actively malicious service providers who can create fake key-pairs for their users without being detected. While LWE splits the MSK on multiple machines, it is not clear if this helps as the lack of software diversity means a successful attack on one machine may be applied to many other machines[34].

3.5 Certificate Transparency (CT)

Certificate Transparency (CT)[36] uses tamper evident logs[24] to build an auditable history of all certificates issued by TLS certificate authorities. The goal of CT is to deter CAs from engaging in impersonation attacks by making such attacks evident in the log. A victim can discover a fake binding for herself by auditing the log.

Tamper evident logs can be securely extended with new certificates, satisfying the *Append-only* property discussed in §2.4. They support membership proofs, allowing users to verify that a certificate is indeed present in the log. However, they do not support non-membership proofs: users cannot efficiently check whether a fake binding exists for them. As a result, expensive *monitoring* has to be performed by

users, who must download and inspect the entire log in order to check that no fake binding has been inserted. This monitoring can be in the realm of practicality for TLS certificates and website owners, since the number of TLS certificates is only around the millions[10, 13]. However, such monitoring is not practical for email users whose number is in the billions[31]. To deter log servers from equivocating about the latest version of the log, CT relies on users and auditors exchanging the commitments of the logs they see with each other via a *gossip protocol*[19]. However, even though the gossip protocol is critical for security, it is not specified in the CT proposal[36].

Certificate Transparency is a promising technique for preventing impersonation attacks, but its lack of efficient non-membership proofs and reliance on an unspecified gossip protocol makes it unusable in the email setting.

3.6 Enhanced Certificate Transparency (ECT)

Enhanced Certificate Transparency (ECT)[47] extends Certificate Transparency (CT) (see §3.5) with non-membership proofs and more efficient monitoring capabilities. ECT uses a binary search tree (BST) called a *LexTree* to map a user's identity to their public key bindings and a tamper evident log[24] similar to CT[36] called a *ChronTree*, which keeps track of the changes in LexTree.

LexTree is a BST sorted by user identities which effectively maps a user's identity to their public key binding (i.e. "Alice" $\rightarrow$ list of last N PKs for Alice). LexTree can be used to provide non-membership proofs. ChronTree is a tamper evident log that logs each addition to LexTree (i.e. $\langle$Alice, PK_{Alice}, root hash of LexTree after adding $PK_{\text{Alice}}\rangle$).

Similar to CT, ECT can efficiently prove the append-only property of ChronTree using consistency proofs as explained in [36, 24]. Additionally, ECT provides a monitoring mechanism by which users can collectively check that the correspondence between ChronTree and LexTree is correct. A user randomly picks a leaf l_i in ChronTree and extracts the binding $\langle \text{user}, PK_{\text{user}}\rangle$ as well as the hash h_i of the LexTree obtained after adding the binding to it. She also extracts the LexTree hash h_{i-1} from the

previous ChronTree leaf l_{i-1}. Next, the user simulates adding PK_{user} to the LexTree committed by h_{i-1} and she checks that the newly obtained LexTree is committed by h_i.

To prevent equivocation about ChronTree and LexTree, ECT assumes a gossip protocol exists by which users can exchange ChronTree and LexTree commitments. However, unless the gossip protocol can guarantee integrity, ECT remains vulnerable to equivocation. Finally, because ECT only stores the last N public keys of a user in LexTree instead of storing the full history, it has a vulnerability by which a user can be impersonated "in plain sight," without efficient detection. However, this can be easily remedied[23].

3.7 CONIKS

CONIKS [40] implements a key server that satisfies the properties described in §2.4. CONIKS uses authenticated dictionaries [25] to store public key bindings, similar to ECT.

An authenticated dictionary in CONIKS is implemented as a Merkle-ized prefix tree. The internal nodes of the tree store cryptographic hashes of their subtrees. The leafs store PK bindings of the form $\langle VUF(\text{user}), PK_{\text{user}} \rangle$, where $VUF(\cdot)$ is a *verifiable unpredictable function*[41] (VUF) used to hide the identities of users in the tree. The root of the tree is signed by the CONIKS key server, creating a small-sized non-repudiable commitment of the server to all the PK bindings in the tree. This is referred to as a *root commitment* and satisfies the *Commitment efficiency* requirement.

To find Alice's PK binding, the binary representation of $VUF(ID_{Alice})$ is used to traverse the tree from the root, down to the leaf where the binding is stored (i.e a zero indicates the binding is in the left subtree; a one indicates it is in the right subtree). A membership proof for Alice consists of a leaf $l = \langle VUF(ID_{Alice}), PK_{Alice} \rangle$ and the hashes of the neighbouring nodes along the path to that leaf l (including the hash of l's neighbouring leaf). The proof can be verified against the root commitment by hashing up from the leaf l until a root hash is obtained. If the obtained root hash is

the same as the hash in the root commitment, then the membership proof is correct[1] (i.e. the key server is not lying about the contents of the leaf l). A non-membership proof for Alice is identical, except that there could either be no leaf found or the found leaf will contain $VUF(ID_{\text{user}}) \neq VUF(ID_{\text{Alice}})$, which proves to the client that Alice does not have a binding in the tree. (Non)membership proofs satisfy the *Query correctness* and the *Query efficiency*[2] requirements.

To provide the *Append-only* property, CONIKS divides time into epochs and generates a new tree every epoch. The new tree includes all items from the old tree plus any new public keys that users registered or updated. The root of a new tree is *appended* to a hash-chain of all previous tree roots. The head node of the hash-chain is signed and becomes the new *root commitment*: a commitment of the current tree's root and of all the previous tree roots, linked together in a hash-chain. Suppose equivocation occurred in epoch n. Then, Alice is presented with a new tree T'_n, different than Bob's tree T_n. As a result, two different hash-chains will be created. The head node of Alice's hash-chain will commit T'_n, while the head node of Bob's hash-chain will commit T_n. Since nodes can only be appended to the hash-chain and the hash function used is collision resistant, Alice and Bob will disagree on all future root commitments. Thus, CONIKS satisfies the *Fork consistency* requirement.

When the CONIKS key server creates the new tree, it promises to include everything from the previous tree, but users do not trust this promise and instead verify the new tree. Every time a new tree is announced, each user verifies via a membership proof that their own public key has stayed the same or has been updated as they requested. This process is called *monitoring*. Note that a user will not be able to (efficiently) tell if other users' public keys have been removed in the new tree[3]. Moreover, a user cannot know if other users' updated public keys are fake or not, so he has to trust that every other user in the system will monitor their own bindings and will whistleblow if they detect impersonation. Having users efficiently monitor their own public key deters the

[1]Note that the path to the leaf needs to correspond to the binary representation in the VUF.

[2]All paths will have $O(\log n)$ nodes because mapping with a VUF will balance the tree.

[3]Consistency proofs for append-only authenticated dictionaries remain an open problem (see §8.4).

CONIKS key server from misbehaving, satisfying the *Append-only* requirement.

Finally, to prevent equivocation and freshness attacks, instead of having a global CONIKS key server responsible for all users from all service providers, each service provider (Google, Microsoft, etc.) runs their own CONIKS key server. Each SP commits itself only to the history of their own users' public keys. Next, all CONIKS key servers participate in a user-verifiable *gossip protocol* with each other to prevent each other from equivocating. Every time a new tree is computed by a key server, the new root commitment is gossiped to all other key servers in the system. When a user receives a new root commitment, he checks it against a random subset of other CONIKS key servers. If the other CONIKS key servers have not received this new root commitment, then it is likely (with the exception of network delays) that the key server is equivocating. Thus, the user needs to wait until it can confirm that root commitment. If the other CONIKS key servers have received a different root commitment for the same epoch, then an equivocation attack is taking place. The two different commitments for the same epoch constitute non-repudiable, globally verifiable, cryptographic proof of equivocation. Under reasonable assumptions about key server collusion, this gossip mechanism satisfies the *Freshness*, *Non-equivocation* and *Proof of misbehaviour* requirements in §2.4.

For a CONIKS deployment to be secure, many service providers need to deploy their own CONIKS servers and gossip with each other. This breaks our *Deployment flexibility* goal (see §2.6). However, we believe that small changes to the CONIKS design can allow providers to deploy CONIKS faster by only deploying a gossip server and outsourcing their key server to another provider they trust.

Unfortunately, CONIKS is vulnerable to coercion and compromise by powerful adversaries as defined in §2.3. Such an adversary can compromise or compel sufficient CONIKS servers and break the gossip protocol, which will lead to equivocation attacks.

3.8 Namecoin

Namecoin[4] is a decentralized cryptocurrency similar to Bitcoin[42], leveraging *blockchains* to build a tamper evident log of transactions. Namecoin was inspired by Bitcoin and shares some of its features such as its proof-of-work mechanism. Unlike Bitcoin, Namecoin is not a monetary currency. Instead, Namecoin was designed to replace the centralized DNS system, enabling users to register $\langle \text{name}, \text{value} \rangle$ pairs by paying for them using the Namecoin currency (i.e. with "namecoins" or NMCs).

Namecoin offers very strong protection against equivocation attacks via its proof-of-work mechanism. Specifically, Namecoin prevents equivocation about the past by storing transactions and $\langle \text{name}, \text{value} \rangle$ registrations in a block-chain protected by proof-of-work, in the same way Bitcoin stores monetary transactions in its block-chain. Additionally, by using *merged mining*[1], Namecoin can use (part of) the power of the Bitcoin network to secure itself.

Unfortunately, as it currently operates, Namecoin is not practical for email security. Users need to download the entire Namecoin block-chain[4] in order to verify that a $\langle \text{name}, \text{value} \rangle$ pair has been registered *and* paid for. This is simply impractical for email users as it would require too much bandwidth both on the users and on the Namecoin network to distribute the block-chain.

3.9 Summary

In this chapter, we presented existing work related to email security and analyzed the strengths and weaknesses in each work. We conclude that none of the existing techniques are resilient to impersonation attacks by powerful adversaries.

[4]The size of the Namecoin block-chain was 2.7GB on August 23rd, 2015

Chapter 4

PowMail architecture

4.1 Overview

PowMail uses an auditable key server design that meets all goals defined in §2.4 and is resilient to attacks by powerful adversaries as defined in §2.3. PowMail key servers employ two data structures to enable users to verify the history of public keys. First, a key-tree similar to the prefix tree in CONIKS[40] is used to verifiably store public key bindings. Key-trees can provide (non)membership proofs when users query for someone's public key. Second, a novel PoW-tree data structure is used to certify the key-tree that users will query. Since key servers can equivocate about the latest key-tree, PowMail employs PoW-trees to enable email users themselves to jointly certify the latest key-tree before they start querying it. PoW-trees are built out of proofs-of-work submitted by email users and are inherently difficult to create, even by powerful adversaries. PoW-trees eliminate the need to gossip key-tree commitments by allowing users to jointly certify the latest key-tree.

Similar to CONIKS (see §3.7), we divide time into epochs and generate a new key-tree that includes all bindings from the previous tree as well as any new bindings that have been registered or updated. A new PoW-tree is also generated every epoch and certifies the previous epoch's key-tree. The PoW-tree generated in epoch n will certify the key-tree generated in epoch $n-1$. PoW-trees give users certainty that the key-tree has been verified by all users and that the key server is not equivocating

about the key-tree. PowMail exposes a simple API for users to register, update and lookup public keys, as illustrated in Table 4.1.

Table 4.1: The PowMail API

API call	**Description**
`Register`(user, PK_{user}, PoW_{user})	Registers a first-time public key
`Update`(user, PK_{user}, PoW_{user})	Updates the user's existing public key
`Lookup`(epoch, user) $\rightarrow$ (PK_{user}, proof)	Looks up the specified user's public key

4.2 Key-trees

PowMail uses Merkle-ized prefix trees called key-trees. Key-trees are similar to the trees used in CONIKS, as described in §3.7. The only difference is that PowMail's key-trees use *path-copying* to save space and computation time when inserting new bindings into the next epoch's tree.

4.3 Proof-of-work

The design of PowMail starts from the observation that proof-of-work can be used to minimize the number of users that an adversary can impersonate. For instance, if a malicious key server wants to perform an *undetected* man-in-the-middle attack (MITM) on Alice and Bob, it needs to create two different trees: one tree for Alice that contains a fake binding for Bob, and one tree for Bob that contains a fake binding for Alice. Similarly, to avoid detection when MITM'ing n victims, n different trees have to be created; one for each user with fake bindings for the other users.

Unfortunately, such attacks are very practical, since it is computationally inexpensive to create a key-tree, as a malicious key server only needs to compute cryptographic hashes and a signature on the root hash. Thus, in order to prevent key servers from equivocating, we seek to make key-tree creation a computationally expensive step. We

can do so by requiring the key servers, the email users, or both to provide some kind of proof-of-work that attests the newly created key-tree. For example, the proof-of-work can be a cryptographic hash that commits the root hash of the key-tree and is smaller than a certain target, achieving a pre-established difficulty level.

Requiring key servers to compute proof-of-work creates the risk of asking for either too little or too much proof-of-work. A motivated and powerful key server could launch an attack unless it is required to produce sufficient proof-of-work. Yet asking for too much proof-of-work could result in the key server being unable to compute it in time for a new epoch. Moreover, asking for too much proof-of-work increases the costs of running a key server, as additional hardware is required. Since it is difficult to make a trade-off between the cost of running a key server and the difficulty of the proof-of-work, we have to turn to the email users for computing the proof-of-work.

Instead of asking a few key servers to compute overly expensive proofs-of-work, we can ask the large number of email users to compute reasonably expensive proofs-of-work. Users will need a way of verifying that enough proof-of-work has been generated for the newest key-tree without resorting to block-chains which are impractical for email (see §3.8).

4.4 Mining incentives

Our system needs to incentivize users to compute proofs-of-work, a process known as *mining*. We think there are a few reasons for which email users would mine:

Mine to register or update public keys. Users can be required to mine before their public keys are registered or updated and they would never use a public key unless it was backed by proof-of-work. The lack of proof-of-work would be a a strong indicator that the public key is fake. Unfortunately, this might not result in generating sufficient proof of work, unless users update their public keys often or new users enter the system frequently, assumptions which are difficult to validate.

Mine to look up public keys. Users can be required to pay for lookups using proof-of-work and would never accept an encrypted email without a fresh proof-of-work

associated with the looked up public key used to encrypt the email. The problem is that a malicious key server can discard these proofs-of-work without being detected because, as opposed to public key registrations and updates which are published in the key-tree, look ups are not published anywhere. Thus, such lookup proofs-of-work would need to be published in a tamper evident log. Note that this could prevent DoS attacks on the key server by making lookups expensive and could also prevent mass encrypted spam, as spammers would need to compute many proofs of work, one for each email address they want to spam. The problem with this approach is that our system loses performance if public key lookups become too expensive. However, this approach could be promising if the difficulty of the lookup proof-of-work is not too high. We plan on investigating this in future work.

Mine to send an email. Users can be required to pay for sending an encrypted email using proof-of-work. Proofs-of-work would be published in a tamper evident log and users would not accept an encrypted email without a proof-of-work. This could also have the advantage of preventing encrypted spam. However, requiring users to mine before sending an encrypted email would affect the performance of our system, breaking our requirements in §2.6.

Mine to keep public key published. Users can be required to pay for keeping their public key published on the key server. This way, mining only needs to be done once per epoch, which could be reasonable for epoch lengths of one day. If a user does not pay, then her public key will be implicitly revoked, preventing her from sending and receiving secure emails. If a public key is not backed by proof-of-work in every epoch, then users have an incentive not to use that public key, as it could have been published by a malicious key server which did not have enough computation power to compute the proof-of-work. As a result, users will not accept or continue to use revoked public keys, as they could be risking their privacy. PowMail adopts this mining incentive since it only requires users to mine once every epoch.

4.5 PoW-trees

As discussed in §4.3, once enough proof-of-work has been generated for a new key-tree, users need a way to verify it without downloading too much data. PowMail uses a novel construction called a PoW-tree which allows users to probabilistically verify that a certain amount of work has been generated for a key-tree. PoW-trees provide a way to aggregate proofs-of-work submitted by users into a single joint proof-of-work that is efficiently verifiable. PoW-trees are different than blockchains. A PoW-tree that aggregates n proofs of work of cumulative difficulty nW can be verified using $O(\log n)$ bandwidth. A blockchain that aggregates n proofs of work needs $O(n)$ bandwidth to be verified. Note that proof-of-work computation can be parallelized for PoW-trees, as each user can compute their own proof-of-work independently of other users. Thus, mining in PowMail is not a competition, as opposed to Bitcoin or Namecoin where users compete to find the proof-of-work for the latest block.

4.5.1 Computing proofs-of-work and creating PoW-trees

Each user will compute a proof-of-work that attests the current key-tree KT_i and is bound to that user's identity. Specifically, a user will compute a cryptographic hash h as follows:

$$h = H(VUF(\text{userid}), H(KT_i), H(PT_{i-1})), \text{such that } h < \frac{\text{max target}}{\text{difficulty}},$$

where $H(KT_i)$ denotes the root hash of the key-tree being attested in epoch i and $H(PT_i)$ denotes the root hash of the previous PoW-tree created in epoch $i-1$.

Depending on the difficulty, the proof-of-work h will require more computation and thus more time to be created. Note that the proof-of-work h contains the root hash $H(KT_i)$ of the newest key-tree as well as the identity of the user who produced it, in the form of a VUF[41] similar to CONIKS[40]. Including the user's identity in the proof-of-work prevents theft and allows the key-server to identify the user who produced the proof-of-work.

To aggregate the proofs-of-work into a PoW-tree, a set of $\langle VUF(\text{user}), h \rangle$ key-value pairs is produced, which maps a user's VUF to her proof-of-work. Next, a *balanced binary search tree* (BST) is built that stores the proofs-of-work in-order, sorted by the user's VUF. We require that the difference in height between any two PoW-tree leafs be no more than one and we give users a way to verify this. This is the same as requiring the tree to be complete, except the last level need not have all leafs as far left as possible. We call such a tree an *almost-complete* tree.

When a new key-tree is published, users will begin certifying it, helping the key server build a PoW-tree for it. Users first verify their binding is valid in the key-tree and will then compute a proof-of-work that certifies the key-tree. Next, users send their proofs-of-work to the key server which aggregates them into a PoW-tree. Once the PoW-tree has aggregated sufficient proof-of-work, it will attest that enough users have verified the key-tree.

4.5.2 PoW-trees as distributed signatures

A PoW-tree can be compared to a distributed signature on a key-tree by all users who have verified their binding in the tree. This signature is unforgeable under the assumption that an adversary will not have sufficient computation power to produce all the required proofs-of-work in the PoW-tree. Note that this *computational assumption* is similar to Bitcoin's assumption. Bitcoin will remain secure (i.e. the block-chain will not be forked) as long as there is no adversary that has more than 50% of the Bitcoin's mining network computation power[42]. PoW-trees solve the gossiping problem: gossip is now implicitly achieved via a valid PoW-tree signature. Furthermore, this gossip channel cannot be easily tampered with, as that would require computing many proofs-of-work. PoW-trees also solve the problem with CONIKS described in §3.7, where if enough CONIKS key-servers were compromised or coerced, then the gossip mechanism in CONIKS would not work anymore and key-servers could equivocate to users without being detected.

4.5.3 Verifying PoW-trees

PoW-trees are verifiable by users, who can check that there is sufficient work in a PoW-tree. A successful check proves that the key-tree signed by the PoW-tree has been verified by enough users, which implies the bindings stored in the key-tree are correct. To ensure that a PoW-tree is a valid signature on a key-tree, users need to verify its:

- **Strength:** Users will check that there are sufficient proofs-of-work in the PoW-tree.
- **Correctness:** Users will check that all the proofs-of-work in the PoW-tree attest the expected key-tree.
- **Difficulty:** Users will ensure that the difficulty of the proofs-of-work in the PoW-tree is sufficiently high.
- **Uniqueness:** Users will ensure that proofs-of-work are not duplicated in the PoW-tree.

Users can validate a PoW-tree by randomly requesting paths in the tree and verifying that:

- The inferred height of the tree is equal to its expected height.
- The difference between path heights is at most one.
- The BST invariants are correct at every node in the paths.
- The proofs-of-work along the paths are sufficiently difficult, commit the expected key-tree and are not duplicated.

The random paths requested by the user are called a PoW-tree *proof*. Note that PoW-tree verification is probabilistic: a user can be certain with high probability that the PoW-tree has a certain strength and commits the expected key-tree. We will analyze the security properties of PoW-tree verification in more detail in §5.2.

4.6 Protocol operation

Below we describe key interactions in the PowMail protocol between users and the key server. We also explain how epochs are created in PowMail.

4.6.1 Epoch creation

An easy way to understand how epochs evolve in PowMail, is to start at the first epoch and detail how public key bindings are added to the system.

When a key server is first deployed, epoch 0 is created which contains an empty key-tree KT_0. The epoch is advertised and users start submitting `Register` requests to the key server with proofs-of-work that certify KT_0. At the end of epoch 0, all the proofs-of-work from the `Register` requests are verified and aggregated in PoW-tree PT_0. Similarly, all the bindings from the `Register` requests are inserted in key-tree KT_1, which will be published at the beginning of epoch 1. Finally, at the end of the epoch, the key server transitions from epoch 0 to epoch 1, publishing KT_1 and PT_0.

In epoch 1, users will not yet be able to query KT_1 as it is not yet certified by a PoW-tree. This is because in epoch 0, the proof-of-work was used to certify KT_0, not KT_1 which was just being built. Thus, users will start certifying KT_1 by submitting proofs-of-work in epoch 1. Users are incentivized to do this; they have to "pay" for keeping their public key bindings in the next epoch's key-tree. Additionally, new users will submit `Register` requests to have their public keys registered in the next epoch's KT_2 key-tree. At the end of epoch 1, all proofs-of-work are verified and aggregated in PT_1. Also, all newly registered bindings as well as any updated bindings are inserted in KT_2. Note that KT_2 will include all the bindings from KT_1, including non-paying users' bindings, even though they will not be valid. The key server transitions from epoch 1 to epoch 2, publishing KT_2 and PT_1. Users can now validate KT_1 by checking that the PT_1 PoW-tree certifies KT_1, as described in §4.5.3. As a result, users will be able to securely perform lookup queries on KT_1. Note that KT_2 will not be ready to be queried until it is certified in epoch 3, when PT_2 is created.

This simple process is repeated during every epoch. At epoch i, the next epoch's

KT_{i+1} key-tree is being built. Concurrently, the PT_i PoW-tree, which certifies epoch i's key-tree KT_i, is also being built.

4.6.2 PoW-tree strength evolution

For PowMail to remain secure, every new epoch's PoW-tree must be at least as strong as the previous epoch's PoW-tree. Otherwise, a malicious key server can create its own PoW-trees that would have insufficient proof-of-work but would be able to attest a key-tree, possibly leading to equivocation attacks. Thus, PowMail needs a mechanism for increasing proof-of-work difficulty and PoW-tree strength as a function of time and as a function of the number of users. Since we expect the number of secure email users to strictly increase over time, PowMail requires new PoW-trees to be at least as strong as previous PoW-trees. Users will verify this is the case and *whistleblow* otherwise. Note that our system's security will increase with the number of users since PoW-trees will accumulate more proof-of-work. To account for Moore's Law, we can double the proof-of-work difficulty once every 18 months. If epochs are one day long, then this will be easy to enforce and verify by the users.

An interesting case could arise where the key server is unable to collect sufficient proofs-of-work from its users for the next epoch. This could happen due to network problems for instance. In this case, the key server should still publish that epoch's PoW-tree but it should indicate in its commitment that the PoW-tree is not strong enough. As a result, users will know the key server is not misbehaving and wait until the next epoch's key-tree is created and certified. Unfortunately, this can lead to an increased waiting time for users who are trying to register or update their public keys, but we do not expect this case to arise often.

4.6.3 Public key registration and update

To register a public key in PowMail:

1. The user generates her key-pair (either a `kex` keypair, a `sig` keypair, or both).

2. The user catches up with the latest epoch n by asking for a non-membership proof in each epoch's key-tree. This proves to the user she has not been impersonated.
3. The user computes a proof-of-work that certifies the key-tree KT_n in the latest epoch n. This will be time consuming, but registration only happens once.
4. The user sends a `Register` request to the key server containing her public key and proof-of-work
5. The key server uses email-based authentication to ensure the user is the owner of the account (see Appendix A.1).
6. The key server ensures the proof-of-work submitted with the request is correct and has enough work.
7. The key server schedules the public key to be included in the next epoch's key-tree KT_{n+1}.
8. The key server saves the proof-of-work in the next epoch's PoW-tree PT_n which will certify the current epoch's key-tree KT_n.
9. The user's public key will be published in the next epoch $n + 1$. However, key-tree KT_{n+1} will not be certified until epoch $n + 2$. As a result, the user's public key will not be visible until then.

Updating a public key operates in the same manner as registration. However, during public key update, a signature under the old `sig` secret key on the new public key is required. Note that this mechanism achieves our *Secure binding update* goal in §2.4, as it only allows the user who registered her initial public key to update it later. This makes impersonation attacks harder for key servers because they will not possess the `sig` secret key of the user. Thus, in order to impersonate a user, a malicious key-server will have to fork the key-tree at an earlier epoch when the victim user was not registered.

4.6.4 Monitoring and public key maintenance

Once the user has registered her public key as described in §4.6.3, she will have to monitor the key server to detect impersonation. Also, the user will have to "pay" for

keeping her public key on the key server by computing proofs-of-work every epoch. The proofs-of-work will certify the key-tree in that epoch, preventing the key server from equivocating.

If the user has registered her public key during epoch n and the key server has included her PK in the next epoch's key-tree KT_{n+1}, then the user will (have to) start monitoring and compute proofs of work once every epoch, starting at epoch $n+1$. For every new epoch $n+i$ where $i > 0$, the user will do the following:

1. The user checks that her binding has remained unchanged in epoch $n+i$ by asking for a membership proof in the KT_{n+i} key-tree.
2. The user computes a proof-of-work that certifies the KT_{n+i} key-tree and sends it to the key server. This step will be time consuming. However, if epochs change once a day, then this can be practical.
3. The user repeats this process for all new epochs.

If the user is offline and misses an epoch, then she will fail to pay for keeping her public key in the key-tree. As a result, the user's public key will be automatically revoked. This is because other users will not use a public key which is not backed by proof-of-work; other users would find it hard to tell whether such a public key is fake or not.

When the user comes back online, she can "catch up" with the current epoch by simply downloading the key-tree in each epoch and verifying via a membership proof that her public key binding has stayed the same. She can then start computing proof-of-work to have her old public key be valid again or she can update her old public key.

4.6.5 Public key lookup

To look up Bob's public key, Alice will do the following:

1. Alice needs to catch up with the latest epoch n, as described in §4.6.4.
2. Alice looks up Bob's PK in key-tree KT_{n-1} by asking for a (non)membership proof for Bob and verifying it.

3. Alice looks up Bob's proof-of-work in PoW-tree PT_{n-1} and verifies it.

Alice can use Bob's public key to send him an encrypted message, as described in §4.7 or she can use Bob's public key to verify his signature on a message, as described in §4.9.

4.7 Message encryption

PowMail uses an elliptic curve public key cryptosystem such as Ed25519[18], allowing users to perform an ECDH key exchange[8] and derive a shared key. The obtained key can be used to encrypt messages between Alice and Bob. Also, the obtained key can be used to deniably authenticate messages between Alice and Bob by computing message authentication codes (MACs). PowMail encrypts email messages using an *authenticated encryption* scheme such as AES-GCM[28].

4.8 Forward secrecy mechanism

To achieve per message forward secrecy, PowMail uses different *message keys* for every email sent, with the exception of the first outgoing email between two users. The first email has to be encrypted with the recipient's public key as users will not have a pre-agreed message key. Message keys are generated independently from users' key-pairs, such that compromise of secret keys does not lead to loss of message secrecy, thereby achieving forward secrecy.

The message keys are agreed upon by piggybacking authenticated Diffie-Hellman key exchanges[27] on top of every email exchanged between users. This way, a new message key can be derived for each email sent and used to encrypt the next email, continuing the cycle. Again, note that the first email message sent from Alice to Bob will not have a pre-agreed message key, and thus will not have forward secrecy. However, cautious users can still protect their secrecy by initially sending a dummy message to the recipient and waiting for a reply, which will implicitly perform the authenticated key exchange. The reply message will be encrypted with a newly

generated message key and will thus have forward secrecy. From that point on, all messages exchanged between the two users will have forward secrecy. We plan on investigating ways to provide forward secrecy for the first email message in future work (see §8.3).

4.9 Message signing

PowMail can enable users to perform non-repudiable digital signatures by allowing them to store different key types on the key server. Users already have a `kex` key-pair that will be used for establishing shared keys using ECDH but using the `kex` secret key for performing digital signatures is ill-advised and could lead to security issues[38]. For instance, an adversary might be able to use peculiarities in the key-exchange protocol to trick the victim into signing arbitrary messages with her secret key. Thus, we introduce a new `sig` key-pair type that can be used to sign and verify messages. Alice can sign a message to Bob using her `sig` secret key and Bob can verify Alice's signature by looking up her `sig` key on the key server.

4.10 Summary

In this chapter, we presented PowMail's architecture. We explained how we use proof-of-work to certify key-trees, obtaining consensus on the public key history in PowMail. We enumerated the mining incentives of email users and argued that the need to keep public keys valid can be used as a strong mining incentive in PowMail. We introduced our novel PoW-tree construction and compared PoW-trees to a distributed unforgeable signature. We explained how PoW-trees can be externally verified by all email users and how they can be used to bootstrap users' trust on the latest commitment of the public key history.

Chapter 5

PowMail security analysis

5.1 Key-tree attacks

Hiding a fake binding. A key server can try to create a fake binding for Alice and hide it next to her real binding. One way to do this would be to use a non-deterministic VUF that can produce different outputs for the same input. In this case, Alice's real binding can be mapped to the leaf determined by $v_1 = VUF(r_1, \text{Alice})$ while her fake binding would be mapped to a different leaf $v_2 = VUF(r_2, \text{Alice})$. Note that r_1 and r_2 denote the randomness or coin flips used to compute the non-deterministic VUF. This attack is easily thwarted by using a deterministic VUF such as a deterministic signature scheme.

Another way to hide a binding would be for the key server to place the real binding in the correct leaf while placing the fake binding somewhere along the path from the leaf to the root (or vice versa). This attack only works if users incorrectly verify the authenticated path returned by the key server. This attack can be detected by Alice when she monitors her binding. The key server will have to show Alice either the real leaf at the end of the path or the fake leaf embedded in the middle of the path. If Alice is shown the real leaf then, when she verifies by hashing up from it, she will obtain an incorrect key-tree root hash due to the extra fake leaf embedded along the path. If Alice is shown the fake leaf only, she will immediately detect impersonation since her real public key is not in that leaf.

Finally, the server can try embedding the fake binding in a wrong leaf (i.e. a leaf that is different than $v = VUF(\text{Alice})$). Again, this only works if users incorrectly verify (non)membership proofs. Users can easily detect the wrong leaf has been returned by verifying that the path to the leaf (i.e. the left and right directions that the path takes) matches the binary representation of the VUF of Alice (i.e. zeros correspond to taking a left pointer, ones, to taking a right pointer).

Equivocation. A key server can try equivocating to users about Alice's binding. The key server can present a new key-tree KT_{Alice} to Alice, which contains her real binding, and present a different key-tree KT_{Others} to the rest of the users. To be able to equivocate, the key server has to produce two PoW-trees: one for KT_{Alice} and one for KT_{Others}. However, producing a PoW-tree requires having as much computation power as the users, which we assume key servers do not have. A key server can equivocate about the key-tree, asking half of its users to produce a PoW-tree for KT_{Alice} and the other half for KT_{Others}. However, the key server will still have to compute half of the remaining work for each PoW-tree or find a way to inflate the PoW-tree with fake work. If we assume that the key server's computational power does not exceed 50% of the users' power, the key server will be unable produce a sufficiently strong PoW-tree. If the server tries to fake the proof-of-work, it will be detected when users validate the PoW-tree (see §5.2).

5.2 PoW-tree attacks

In this section we describe attacks that try to spuriously *inflate* the amount of work in a PoW-tree, in order to forge a signature on a key-tree. Our main goal is to prevent the attacker from creating fake PoW-trees that have a significant fraction f of fake proofs-of-work. Recall our threat model assumption that a malicious key server's computation power will not exceed 50% of the computation power of its users. Thus, the fraction f of fake proofs-of-work will typically be close to $\frac{1}{2}$. When users detect an attack, they will *whistleblow* and announce misbehaviour to everyone.

5.2.1 Reused and bogus proofs-of-work

An attacker might try to inflate the work in a PoW-tree by inserting proofs-of-work from previous PoW-trees or bogus proofs-of-work which commit the wrong key-tree hash or the wrong VUF or which are not sufficiently difficult. For now, we assume the attacker inserts the bogus proofs-of-work in the tree correctly, respecting BST invariants, and deal with the case where the attacker breaks BST invariants in §5.2.2. Reused proofs-of-work will be quickly found in the PoW-tree when users randomly query it and, when verified, a wrong key-tree root hash will be detected which proves key server misbehaviour. Bogus proofs-of-work that have a smaller-than-required difficulty or commit the wrong key-tree hash or wrong VUF will be detected in the same fashion.

Proof sketch. Recall that PoW-trees are balanced binary search trees that store $\langle VUF(\text{user}), PoW_{\text{user}} \rangle$ pairs, sorted by $VUF(\text{user})$. Suppose the malicious key server claims there are n proofs-of-work in the tree, but a fraction f of them are bogus. First, note that by asking for a random node in the tree, a bogus proof-of-work will be found with probability f. Second, note that by asking for a random path in the tree, a bogus proof-of-work will be found with probability $\Pr[\text{bogus } PoW \text{ found}] > f$. Thus, the probability of not finding a bogus proof of work after asking for one random path is $\Pr[\text{bogus } PoW \text{ not found}] < 1 - f$. Third, note that by asking for k random paths in the PoW-tree, the probability of not finding a bogus proof becomes arbitrarily small:

$$\Pr[\text{bogus } PoW \text{ not found after } k \text{ queries}] < (1 - f)^k$$

5.2.2 Duplicating proofs-of-work

Since reused and bogus proofs-of-work can be detected, an attacker might try duplicating valid proofs-of-work in the same PoW-tree. Recall that the PoW-tree is a BST sorted by the user's VUF. As a result, an attacker cannot insert a duplicate proof-of-work without breaking the tree's BST invariant at some node. The attacker can try inserting the duplicate proof-of-work under a different user's VUF, but this

will be detected as a bogus proof of work (see above). We show below that BST invariant violations will be quickly detected when users randomly verify the PoW-tree.

Proof sketch. Suppose the key server claims there are n proofs-of-work in the tree, but a fraction f of them are duplicated and, as a result, break BST invariants. Recall that a BST invariant violation is a pair of any two nodes along a path that are not properly ordered. First, notice that each duplicate proof-of-work inserted will have to increase the number of BST invariant violations by (at least) one. Second, notice that each BST invariant violation creates one or more invalid paths in tree (i.e. a path containing two nodes that are out of order). For instance, if the violation is between the root of the tree and a leaf of the tree, then the violation creates a single invalid path. However, if the violation is between the root of the tree[1] and a non-leaf node, then the violation creates multiple invalid paths; as many as the number of leafs below the non-leaf node. It can be shown that the number of invalid paths created is related to the number of violations and hence the number of duplicate proofs-of-work inserted.

$$\text{no. of invalid paths} \geq \frac{\text{no. of BST invariant violations}}{2} \Rightarrow$$
$$\text{no. of invalid paths} \geq \frac{\text{no. of duplicate proofs-of-work}}{2}$$

Thus, since a fraction f of nodes contain duplicate proofs-of-work, it follows that a fraction $f/2$ of paths will be invalid. Then, as explained in §5.2.1, the probability of not finding a duplicate proof-of-work will be:

$$\Pr\left[\text{duplicate } PoW \text{ not found after } k \text{ queries}\right] = \left(1 - \frac{f}{2}\right)^k$$

Finally, we can decrease the failure probability to $(1-f)^k$ during verification by including the proof-of-work in the neighbours of the nodes in the PoW-tree proof, instead of only providing their hashes. This will enable detection of BST violations between the nodes along a path and their neighbours, effectively doubling the number

[1] Any node below the root can suffice though.

of invalid paths.

5.2.3 Bogus tree height

We have shown that increasing the tree's height by inserting a fraction f of duplicate, reused or bogus proofs-of-work fails with high probability. Thus, the adversary might try rearranging the tree by increasing its height to make it appear larger, but without inserting any proofs-of-work. As a result, the adversary will create empty subtrees in the PoW-tree, resulting in a tree that is not almost-complete as described in §4.5.1. We show that such empty subtrees can be easily detected, proving misbehaviour.

Proof sketch. We can reduce this case to the previously examined case where the adversary inserts bogus proofs-of-work. Note that the empty subtrees that are created after the height of the tree is artificially increased are equivalent to bogus proofs-of-work subtrees inserted in a PoW-tree to make it appear larger. Using a similar argument as in §5.2.1, an adversary who wants to add a fraction f of fake proofs-of-work by artificially increasing the PoW-tree's height will succeed with negligible probability:

$$\Pr[\text{bogus PoW-tree height not detected after } k \text{ queries}] = (1-f)^k$$

5.3 Key server security properties

First, since PowMail uses prefix Merkle trees that are monitored by users, much like CONIKS[40] does, it achieves all goals described in §2.4, except for *Non-equivocation*. Second, PowMail's novel use of PoW-trees enables it to meet the *Non-equivocation* goal in the face of powerful adversaries, as defined in §2.3. Third, the *Freshness* goal can be met by assuming users have loosely synchronized clocks and can thus tell when a new key-tree and PoW-tree should be published.

As we detailed in this chapter, PoW-trees are employed to certify a key-tree. Creating a PoW-tree requires computing many proofs-of-work, which a powerful adversary will not be able to do without investing a large amount of capital in an attack. Furthermore, the adversary's cost will double with the number of forks he

wants to maintain, as each fork has a different key-tree, which requires a different PoW-tree to certify it. We believe the high costs associated with creating forks will prevent attackers from equivocating. Thus, PowMail's key server design achieves all properties described in §2.4.

5.4 Email transmission security properties

PowMail's secure key servers enables users to correctly obtain each other's public keys and to send and receive emails with all properties described in §2.5. As a result, PowMail can easily achieve its security goals for email transmission:

- *Deniable authentication* is achieved by allowing users to exchange a shared secret key via ECDH[8] (see §4.7).
- *Secrecy and integrity* is achieved by encrypting using an authenticated cipher mode under the shared secret key (see §4.7).
- *Forward secrecy* is achieved by exchanging message keys over email communication using piggybacked Diffie-Hellman key-exchanges (see §4.8).
- *Non-deniable authentication* can be achieved by signing messages using the `sig` key-pair type (see §4.9).

5.5 Summary

In this chapter, we detailed attacks that key servers can attempt. We explained how key-tree attacks are easily detected and thwarted by correctly verifying (non)membership proofs. For PoW-tree attacks, we showed that under a reasonable computational assumption for key servers, our system remains secure. We conclude that PowMail achieves all of its key server security goals as well as its email transmission security goals.

Chapter 6

PowMail implementation

We used the Go programming language to implement a key server and a PoW server in 3500 lines of code. The key server manages the key-tree: it allows users to register, update and verify their bindings, it enforces the mining of proofs-of-work by rejecting public key registration or update requests unless they include the required proof-of-work. The PoW server manages the PoW-tree: it collects the proofs-of-work submitted to the key server, it creates PoW-trees and replies to queries from users that verify the PoW-tree. Note that the PoW server could also be implemented within the key server. We chose to separate their implementations to increase the key-tree and PoW-tree query throughput by having services run on different machines. Below, we describe some of the details of our implementation.

Protocol Buffers. We used the Protocol Buffers[6] library built by Google to implement tree path serialization for key-tree (non)membership proofs and PoW-tree proofs.

gRPC. We used the gRPC library to implement the key server and PoW server as RPC servers. The gRPC library has the advantage of not having to rewrite client RPC code when writing PowMail clients for different platforms. For instance, a JavaScript client would be required for a webmail implementation of PowMail, while a C++ or Java implementation would be needed for porting existing mail clients to use PowMail. Using gRPC, we can generate RPC client code automatically for any language we write a PowMail client in.

Path-copying. We used path-copying to save memory when new key-trees are created. This speeds up insertion in the key-tree and allows PowMail to create epochs instantly without cloning the previous epoch's key-tree.

6.1 Key server API

The key server exposes a simple API which allows users to obtain the root commitments of key-trees, register public keys and update public keys.

Table 6.1: The key server API

API call		**Return type**
`GetCommit(epoch)`	$\rightarrow$	$\langle Commit(KT_{\text{epoch}}), Commit(PT_{\text{epoch}})\rangle$
`LookupPk(epoch, user)`	$\rightarrow$	$\langle PK_{\text{user}}, KT_{proof}, PT_{proof}\rangle$
`RegPk(user,` PK_{user}, PoW_{user}`)`	$\rightarrow$	Signed promise to include PK_{user}
`UpdPk(user,` PK_{user}, PoW_{user}`)`	$\rightarrow$	Signed promise to update user's PK

6.2 PoW server API

The PoW server exposes a simple API for queuing proofs-of-work to be added to the next PoW-tree, for verifying PoW-trees and for finding proofs-of-work submitted by users.

Table 6.2: The PoW server API

API call		**Return type**
`AddPow(user,` PoW_{user}`)`	$\rightarrow$	Signed promise to include PoW_{user} in the next PoW-tree
`Verify(user, rand)`	$\rightarrow$	PoW-tree paths as selected by `rand` coin-flips
`FindPow(epoch, user)`	$\rightarrow$	$\langle PoW_{\text{user}}, PT_{proof}\rangle$

6.3 Summary

In this chapter, we described our Go implementation of PowMail. Our prototype exposes a simple interface which clients can use to register, update or look up public keys.

Chapter 7

PowMail evaluation

7.1 Key-tree lookups

For this experiment, we set up a key-tree with $2^{24} = 16777216$ bindings and we measured the rate at which the key server could compute (non)membership proofs for public key lookups. We then measured clients' (non)membership proof verification time. We did not measure the client signature verification time for the (non)membership proof since the client can simply compare the root hash in the proof against the root hash in its latest key-tree commitment, whose signature it has verified at the beginning of the epoch. In both experiments, we sampled by executing 65,536 `LookupPk` queries. The results can be seen in Figure 7-1 and 7-2.

Most client verification times fell within 5-10 μseconds, with a 7.5 μsecond average and 3.2 μsecond standard deviation. Most server proof computation times fell within 20-40 μseconds, with a 20.1 μsecond average and 3.2 μsecond standard deviation. These results indicate that the key server can handle up to $\frac{1}{20.1/10^6} \approx 50{,}000$ proofs per second, the equivalent of 4.3 billion proofs per day. Note that since an epoch's key-tree does not change after it is created, we can improve server proof throughput by using multiple threads to generate proofs. Finally, we can further increase throughput by replicating the read-only key-tree on multiple machines.

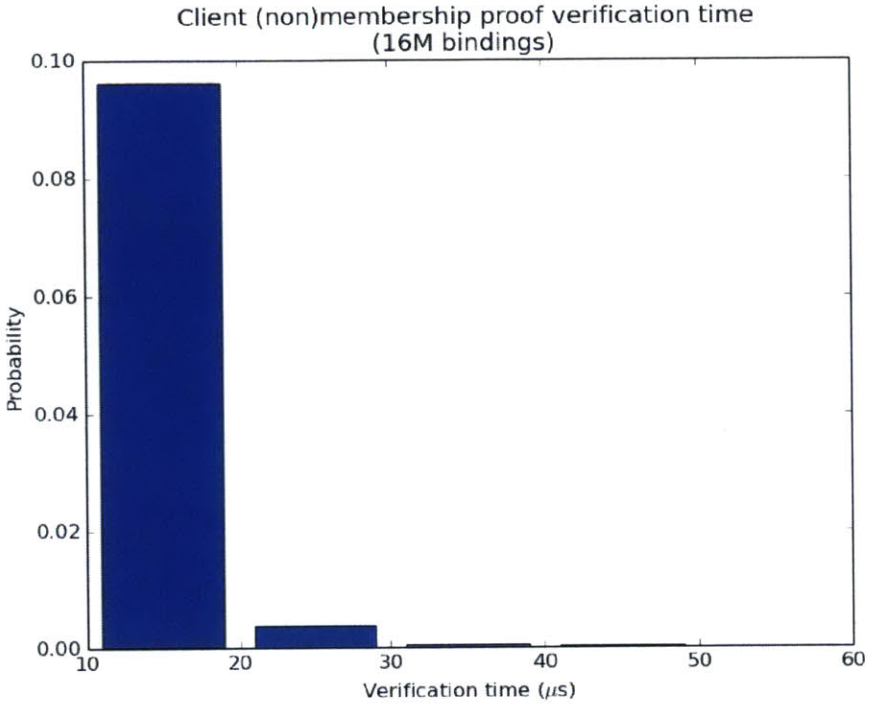


Figure 7-1: Client (non)membership proof verification time histogram

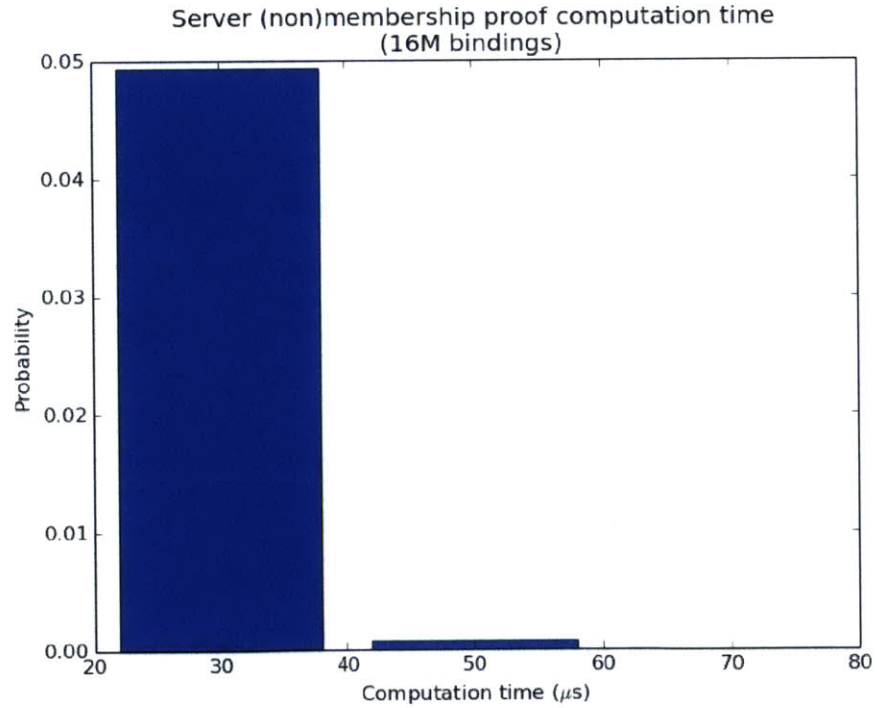
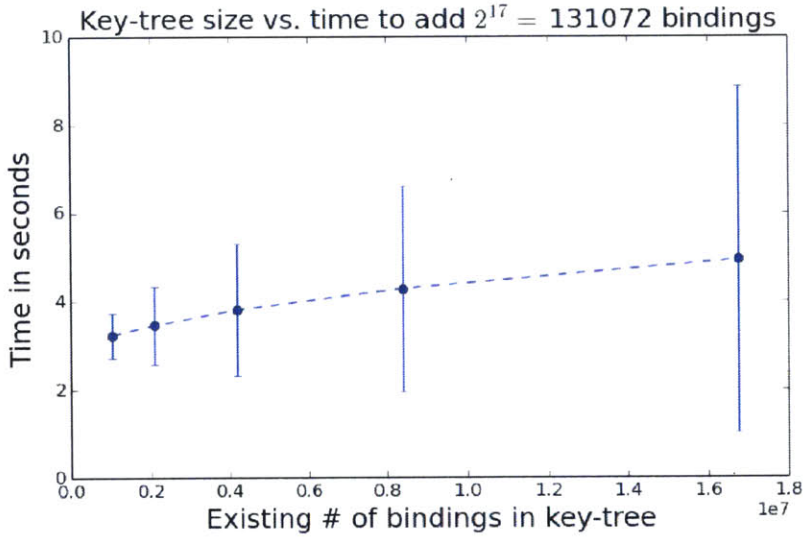


Figure 7-2: Server (non)membership proof computation time histogram

7.2 Key-tree new epoch computation

For this experiment we measured the time to compute new key-trees for new epochs. We setup an initial key-tree with 2^{20} bindings. Next, we added a batch of 2^{17} new bindings to the key-tree and measured the server insertion time. We then repeated the experiment with key-trees of size $2^{21}, 2^{22}, 2^{23}$ and 2^{24} bindings. We sampled 50 executions for each key-tree size. The results can be seen in Figure 7-3.



[h]

Figure 7-3: Server time to insert 131,072 bindings as a function of key-tree size

Our results show that insertion time increases slowly despite doubling the key-tree size for each point. Our insertion performance is higher than CONIKS because we use path-copying when creating a new key-tree, avoiding the need to clone the tree when a new epoch is generated.

7.3 PoW-tree construction

For this experiment we measured the time to build a PoW-tree as a function of the number of proofs-of-work to be aggregated in the tree. We doubled the PoW-tree size for each test, starting with $2^{20} = 1048576$ proofs-of-work and going up to $2^{24} = 16777216$ proofs-of-work. We then measured the time it took the PoW server to compute the balanced binary search tree. The results can be seen in Figure 7-4.

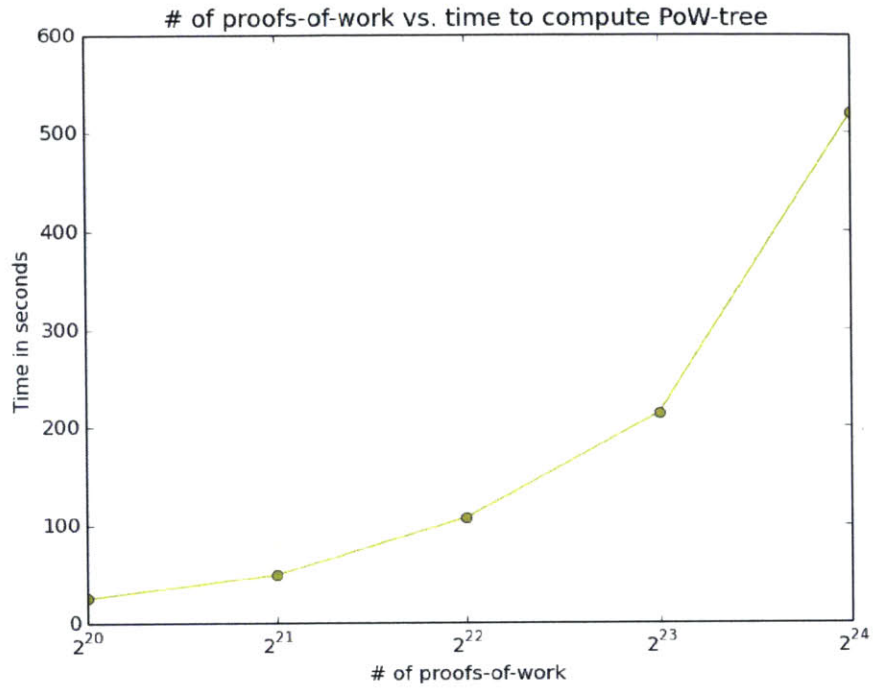


Figure 7-4: PoW-tree construction time as a function of its size

Since we are sorting the proofs-of-work by their users' VUF and creating a balanced BST, the time to create a PoW-tree is dominated by the time to perform the sort.

7.4 Summary

In this chapter, we evaluated the performance of our system. We showed that key servers can support 50,000 key-tree lookup queries per second and can be easily scaled using replication and concurrency. Also, we showed that path-copying key-trees insert performance can scale easily. Finally, we showed that PoW-tree construction is the most expensive step and it should be further optimized.

Chapter 8

Future work

8.1 Webmail

Webmail is difficult to secure since providers such as Gmail or Hotmail could easily extract secret keys from users' browsers by serving modified JavaScript code to their users. For example, users' trust in ProtonMail[7], a secure webmail provider, could be misplaced not only because ProtonMail could undetectably misbehave but also because users are required to trust all TLS certificate authorities not to impersonate ProtonMail and inject modified JavaScript that reads users' secret keys. We propose two possible solutions to the secure webmail problem.

The first (and preferred) solution involves dramatically redesigning browsers to verify encrypted web applications such as secure webmail. Browsers should be re-designed to verify that webserver-provided JavaScript code abides by a security policy specified via a domain-specific language in a *trusted policy file*. Browsers can pin such policy files and ask the user to inspect policy updates[1]. Ideally, the domain-specific language used to specify the policy should be user-comprehensible. Furthermore, to prevent equivocation about the policy file, a *policy transparency* technique similar to Certificate Transparency[36] could be used.

The second solution is for users to use a trustworthy browser extension. Instead of trusting a web-mail provider, users can trust a browser extension to safely store their

[1]Of course, leaving security decision to users can be ill-advised and would need careful consideration.

secret keys and secure their webmail communications by transparently encrypting and decrypting emails within browser pages. Such an extension can be verified by third parties to ensure it does not leak user's secret keys or their data. We plan on addressing secure webmail in future work.

8.2 Fighting (encrypted) spam

Email encryption can be a spammer's paradise, exacerbating the spam problem by enabling spammers to obfuscate their emails using encryption and avoid detection on the service provider side. However, we believe our proof-of-work mechanism for registering public keys coupled with email authentication can deter large-scale spamming. If botnets did not exist, then a spammer using a single account for sending encrypted spam could be easily stopped: users simply report the spam email and the spammer's account is flagged by their provider. As a result, spammers would have to register many accounts which would require computing expensive proofs-of-work thereby making spam cost-ineffective.

Since botnets are a reality, with over 1.9 million bots on the internet[50], they need to be addressed. A spammer can use a bot to create one or more spam accounts by stealing the victim's computation power and computing the required proof-of-work to register and maintain the public key bindings in PowMail. The spammer can then send encrypted spam from these accounts, but, as discussed before, such accounts would be easily reported by users and flagged as spam accounts. Also, the high CPU usage associated with computing the proof-of-work could be noticed by some users who would suspect their machines are compromised. To avoid being detected, spammers can find the secret key of the victim's PowMail account and use the victim's account instead of creating new accounts, avoiding the need to consume the victim's CPU and minimizing detection risk. Such spam emails would be authenticated, making the email appear legitimate to the recipients, further increasing the effectiveness of the spam. We believe this kind of spam could be detected or at least throttled at the service provider (SP) level. For instance, service providers could deploy spam

filters that use machine learning over encrypted data algorithms, as proposed in recent work[21]. Another approach would be for service providers to monitor the number of emails sent by their users in order to detect potentially compromised accounts. For this to work, a service provider should only accept encrypted email with a signature by the sender's service provider who has verified that the sender's account is not compromised. This could be effective, since the number of bots is in the millions while the number of spam emails is in the tens of billions[50], meaning bots need to send thousands of emails from individual accounts which would send a loud signal to service providers that an account is compromised[2].

While requiring proof-of-work for sending an email has been shown infeasible for preventing spam due to spammers ability to control end-user machines[35], we believe requiring proof-of-work for account registration could be more effective, as it could prevent spammers earlier by limiting the number of accounts they can use. We plan on addressing spam botnets in future work by taking advantage of the implicit authentication in secure email to detect and throttle spam, hopefully making it cost-ineffective.

8.3 Forward secrecy for the first email

Forward secrecy for the first email message can be hard to obtain since users will not agree on message keys until the sender gets a reply to the first email from the receiver. To provide forward secrecy for the first email message, we can provide an option in the user interface that enables a user to start a message key exchange with one of her contacts without sending a *dummy* email and waiting for a dummy reply (see §4.8). Such a feature can be implemented transparently so that no additional emails will be displayed in the communicating users' inboxes. A different route would be to introduce a special kind of *forward-secure email conversation* in which, before sending the first email in the conversation, a transparent key-exchange is performed by sending

[2]Multiplexing compromised accounts will not give bots any advantage as the number of emails seen and thus filtered out by SPs will remain the same.

a *request key-exchange email* which prompts the recipient to reply with an *accept key-exchange email*, bootstrapping an authenticated Diffie-Hellman key-exchange[27]. When the sender receives the *accept key-exchange email*, he sends the actual first email, encrypted under the exchanged message key. Note that the key-exchange emails would be sent and handled transparently by PowMail clients, not creating any overhead for the user. The disadvantage of this method is that email delivery will not happen instantly. If Alice sends the first email in a forward-secure conversation to Bob and then goes offline for a year, then Bob will not receive the email until a year later when Alice comes back online, finishes the key-exchange and sends the actual email to Bob. We plan on addressing forward secrecy for the first email in future work.

8.4 Append-only authenticated dictionaries

To prevent key servers from equivocating about the history of its users' public keys, an *append-only authenticated dictionary* is needed along with a gossip protocol, which prevents the key server from equivocating about the dictionary itself. While this thesis addresses the need for a gossip protocol by introducing PoW-trees to certify the latest dictionary, the append-only authenticated dictionary problem remains unsolved. Note that CONIKS is not an append-only dictionary because, while users can ensure their own history of bindings has remained append-only, they cannot ensure that the history of other users' bindings has remained append-only. However, the monitoring done by all users, the dividing of time into epochs and the CONIKS gossip protocol together provide the append-only property for all *monitoring* CONIKS users. Previous work such as ECT[47] and Balloon[46] also try to solve the append-only authenticated dictionary problem but they can only enforce the append-only property by requiring users to monitor the data structure, which can be expensive. The problem of providing logarithmic-sized consistency proofs in append-only authenticated dictionaries remains an open problem which we will explore in future work.

8.5 Whistleblowing

In our security analysis (see §5), we mentioned that users will *whistleblow* when they detect an attack, without fully specifying how this will happen. We believe that enabling users to whistleblow and making this *evident* to other users will provide an important health metric for PowMail's security and we plan on addressing it in future work.

8.6 Summary

In this chapter, we proposed future work to better deal with the challenges of secure webmail, encrypted spam, forward secrecy for the first email and whistleblowing to announce misbehaviour in PowMail.

Chapter 9

Conclusion

In this thesis, we showed that a secure email solution can be built using proof-of-work consensus. We presented PowMail, an architecture for secure email communication that leverages proof-of-work and a novel PoW-tree primitive to prevent impersonation attacks by powerful attackers.

Our contributions were the following:

- We designed, implemented and evaluated PowMail, a secure end-to-end email security architecture.
- We introduced a novel authenticated data structured called a PoW-tree which allows a large number of users to compute a distributed unforgeable signature and reasoned about its security.
- We proposed a security model for key servers in email.
- We summarized promising previous work on secure email and secure public key distribution.
- We proposed directions for future work on secure web-based email, encrypted spam, whistleblowing and forward-secrecy for the first email message.

Appendix A

Discussions

A.1 Initial email-based authentication

Before users allowing users to register a public key, the key server needs to authenticate them via their service provider. Without such authentication, Mallory, an active adversary, could register accounts for many users of the system (as many as he can compute proofs-of-work for) and, as a result, create a denial of service (DoS) attack, since these users will not be able to override the fake public keys published by Mallory without knowing their corresponding secret keys. To authenticate Alice's `Register` request, the key server can email Alice and verify her ownership of the account by having Alice click on a unique link stored in the email[29]. However, since active attackers can intercept that (unencrypted) email, the key server needs a way of authenticating Alice's service provider and asking it to deliver the email to Alice. This can be achieved via the existing TLS infrastructure with the caveat that this initial email-based authentication remains secure only if all TLS CAs behave correctly.

Bibliography

[1] Bitcoin merge mining specification. https://en.bitcoin.it/wiki/Merged_mining_specification. Accessed: 2015-08-23.

[2] Convergence. http://www.convergence.io/. Accessed: 2015-08-22.

[3] MIT PGP Public Key Server. https://pgp.mit.edu.

[4] Namecoin. https://namecoin.info/. Accessed: 2015-08-23.

[5] Perspectives project. http://perspectives-project.org/. Accessed: 2015-08-22.

[6] Protocol Buffers. https://developers.google.com/protocol-buffers/?hl=en.

[7] ProtonMail. http://protonmail.ch.

[8] SEC 1: Elliptic Curve Cryptography. http://www.secg.org/sec1-v2.pdf, may 2009.

[9] A new approach to China. http://googleblog.blogspot.com/2010/01/new-approach-to-china.html, jan 2010. Accessed: 2015-08-24.

[10] Internet SSL Survey 2010 Results. https://community.qualys.com/blogs/securitylabs/2010/07/30/internet-ssl-survey-2010-results, July 2010.

[11] Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification. RFC 5751, RFC Editor, Jan 2010.

[12] SSL And The Future Of Authenticity. http://www.thoughtcrime.org/blog/ssl-and-the-future-of-authenticity/, April 2011.

[13] January 2014 web server survey. http://news.netcraft.com/archives/2014/01/03/january-2014-web-server-survey.html, January 2014.

[14] Ben Adida, David Chau, Susan Hohenberger, and Ronald L. Rivest. Lightweight email signatures (extended abstract). In *Proceedings of the 5th International Conference on Security and Cryptography for Networks*, SCN'06, pages 288–302, Berlin, Heidelberg, 2006. Springer-Verlag.

[15] Ben Adida, Susan Hohenberger, and Ronald L. Rivest. Lightweight encryption for email. In *Proceedings of the Steps to Reducing Unwanted Traffic on the Internet on Steps to Reducing Unwanted Traffic on the Internet Workshop*, SRUTI'05, pages 13–13, Berkeley, CA, USA, 2005. USENIX Association.

[16] Heather Adkins. An update on attempted man-in-the-middle attacks. `http://googleonlinesecurity.blogspot.com/2011/08/update-on-attempted-man-in-middle.html`. Accessed: 2015-08-22.

[17] Steven M. Bellovin. Spamming, phishing, authentication, and privacy. *Commun. ACM*, 47(12):144–, December 2004.

[18] DanielJ. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

[19] Ken Birman. The promise, and limitations, of gossip protocols. *SIGOPS Oper. Syst. Rev.*, 41(5):8–13, October 2007.

[20] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-The-Record Communication, or, Why Not to Use PGP. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 77–84, New York, NY, USA, 2004. ACM.

[21] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine Learning Classification over Encrypted Data. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, 2015.

[22] Peter Bright. `http://arstechnica.com/security/2010/03/govts-certificate-authorities-conspire-to-spy-on-ssl-users/`. Accessed: 2015-08-22.

[23] Vincent Cheval, Mark Ryan, and Jiangshan Yu. DTKI: a new formalized PKI with no trusted parties.

[24] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 317–334, Berkeley, CA, USA, 2009. USENIX Association.

[25] Scott A. Crosby and Dan S. Wallach. Authenticated dictionaries: Real-world costs and trade-offs. *ACM Trans. Inf. Syst. Secur.*, 14(2):17:1–17:30, September 2011.

[26] M. Kucherawy D. Crocker, T. Hansen. RFC: DomainKeys Identified Mail (DKIM) Signatures. `http://tools.ietf.org/html/rfc6376`. Accessed: 2015-08-23.

[27] W. Diffie and M.E. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, Nov 1976.

[28] Morris Dworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf, nov 2007.

[29] Simson L. Garfinkel. Email-based identification and authentication: An alternative to pki? *IEEE Security and Privacy*, 1(6):20–26, November 2003.

[30] Doug Gross and Brandon Griggs. Snapchat CEO 'Mortified' by Leaked E-Mails. http://www.cnn.com/2014/05/29/tech/mobile/spiegel-snapchat-leaked-e-mails/. Accessed: 2015-5-14.

[31] The Radicati Group. Email statistics report, 2014-2018. http://www.radicati.com/wp/wp-content/uploads/2014/01/Email-Statistics-Report-2014-2018-Executive-Summary.pdf, 2014.

[32] Jen Heger. Inside Job? FBI Focusing On Sony Employee As Potential Source Of Hacker Leak. http://radaronline.com/exclusives/2014/12/sony-email-leak-inside-job-claim-fbi-interview-staff/. Accessed: 2015-05-14.

[33] Eduard Kovacs. Study: Most Firms Send Sensitive Data via Email, but Fail to Protect It. http://news.softpedia.com/news/Study-Most-Firms-Send-Sensitive-Data-Via-Email-But-Fail-to-Protect-It-Video-27shtml. Accessed: 2015-01-31.

[34] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated Software Diversity. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 276–291, May 2014.

[35] Ben Laurie and Richard Clayton. "proof-of-work" proves not to work. In *IN WEAS 04*, 2004.

[36] Ben Laurie, Adam Langley, and Emilia Kasper. RFC: Certificate Transparency. http://tools.ietf.org/html/rfc6962. Accessed: 2015-5-13.

[37] Ravi Mandalia. Security breach in CA networks - Comodo, DigiNotar, GlobalSign. http://blog.isc2.org/isc2_blog/2012/04/test.html. Accessed: 2015-08-22.

[38] K.M. Martin. *Everyday Cryptography: Fundamental Principles and Applications*. OUP Oxford, 2012.

[39] David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 108–117, New York, NY, USA, 2002. ACM.

[40] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. Bringing deployable key transparency to end users. In *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C., August 2015. USENIX Association.

[41] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *In Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS*, pages 120–130. IEEE, 1999.

[42] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[43] C. Newman. RFC: Using TLS with IMAP, POP3 and ACAP. `http://tools.ietf.org/html/rfc6376`.

[44] Danny O'Brien. Web users in the United Arab Emirates have more to worry about than having just their BlackBerries cracked. `http://www.slate.com/articles/technology/webhead/2010/08/the_internets_secret_back_door.html`. Accessed: 2015-08-22.

[45] Nicole Perlroth. Yahoo Breach Extends Beyond Yahoo to Gmail, Hotmail, AOL Users. `http://bits.blogs.nytimes.com/2012/07/12/yahoo-breach-extends-beyond-yahoo-to-gmail-hotmail-aol-users/?_r=0`, 2012. Accessed: 2015-08-24.

[46] Tobias Pulls and Roel Peeters. Balloon: A forward-secure append-only persistent authenticated data structure. Cryptology ePrint Archive, Report 2015/007, 2015. `http://eprint.iacr.org/`.

[47] Mark D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. Cryptology ePrint Archive, Report 2013/595, 2013. `http://eprint.iacr.org/`.

[48] Bruce Schneier. New NSA Leak Shows MITM Attacks Against Major Internet Services. `https://www.schneier.com/blog/archives/2013/09/new_nsa_leak_sh.html`. Accessed: 2015-08-22.

[49] Adi Shamir. Identity-based Cryptosystems and Signature Schemes. In *Proceedings of CRYPTO 84 on Advances in Cryptology*, pages 47–53, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[50] Symantec. 2015 Internet Security Threat Report, Volume 20. `https://www.symantec.com/security_response/publications/threatreport.jsp`.

[51] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, and Bryan Ford. Decentralizing authorities into scalable strongest-link cothorities. *CoRR*, abs/1503.08768, 2015.

[52] E Rescorla T. Dierks. RFC: The Transport Layer Security (TLS) Protocol, Version 1.2. `https://tools.ietf.org/html/rfc5246`. Accessed: 2015-08-22.

[53] Debra Cassens Weiss. Leaked Emails that Led to Exposure of Petraeus Affair Violated Socialite's Privacy, Suit Claims. http://www.abajournal.com/news/article/leaked_emails_that_led_to_exposure_of_petraeus_affair_violated_socialites_p. Accessed: 2015-5-14.

[54] Alma Whitten and J. D. Tygar. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, SSYM'99, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[55] Philip R. Zimmermann. *The Official PGP User's Guide.* MIT Press, Cambridge, MA, USA, 1995.