ClaimChain: Decentralized Public Key Infrastructure

Bogdan Kulynych[1,2], Marios Isaakidis[3], Carmela Troncoso[1], and George Danezis[3]

[1]IMDEA Software Institute, Madrid, Spain
[2]Polytechnic University of Madrid
{bogdan.kulynych, carmela.troncoso}@imdea.org
[3]University College London, UK
{marios.isaakidis.15, g.danezis}@ucl.ac.uk

July 18, 2017

Abstract

We envision a decentralized Public Key Infrastructure (PKI) design, that we call ClaimChain, where each user or device maintains repositories of claims regarding their own key material, and their beliefs about public keys and, generally, state of other users of the system. High integrity of the repositories is maintained by virtue of storing claims on authenticated data structures, namely hash chains and Merkle trees, and their authenticity and non-repudiation by the use of digital signatures. We introduce the concept of cross-referencing of hash chains as a way of efficient and verifiable vouching about states of other users. ClaimChain allows to detect chain compromises, manifested as forks of hash chains, and to implement various social policies for deriving decisions about the latest state of users in the system.

The claims about keys of other people introduces a privacy problem that does not exist in the centralized PKI design. Such information can reveal the social graph, and sometimes even communication patterns. To solve this, we use cryptographic verifiable random functions to derive private identifiers that are re-randomized on each chain update. This allows to openly and verifiably publish claims that can only be read by the authorized users, ensuring *privacy of the social graph*. Moreover, the specific construction of Merkle trees in ClaimChain, along with the usage of verifiable random functions, ensures users *can not equivocate* about the state of other people. ClaimChain is flexible with respect to deployment options, supporting fully decentralized deployments, as well as centralized, federated, and hybrid modes of operation.

We have evaluated the ClaimChain's computation and memory requirements using a prototype implementation. We also simulated the flow of the system using Enron dataset, comprising real-world email communication history within an organization, in order to evaluate the effectiveness of propagation of key material in a fully decentralized setting.

Contents

1 Introduction

Following the Snowden revelations it became clear that, given the dependence of citizens, governments, and corporations on electronic communications, there is a strong need for highly secure end-to-end communications in which content cannot be accessed by third parties, to shield communications from mass surveillance systems, domestic or foreign.

Content confidentiality in end-to-end communication systems can be achieved by relying on asymmetric cryptography techniques to protect the transmitted data. These techniques work as follows: each user generates a pair of keys, namely the private decryption key, that is kept secret, and the public encryption key that can be shared publicly. Those who know the public key can use it to encrypt messages in such a way that only the owner of the respective private key can decrypt them. Public key cryptography can also be used for authentication and integrity purposes. The owner of the private signature key can digitally sign a message and those who know the public verification key can verify the signature and be assured that (i) the message has not been tampered with, and (ii) the message comes from the singature key owner.

Although asymmetric cryptography solves the confidentiality problem, it requires that users know each others' keys, with high assurance. Public Key Infrastructure (PKI) is a system which facilitates the operations that relate to public key exchange required by secure communications. PKI creates *trusted* bindings between an entity and their respective public keys, and provide means to recording, distributing and revoking public keys. A core requirement for a PKI is therefore to guarantee high integrity of this binding information. Otherwise, if an adversary could confuse an honest user about the binding between other user's identity and public key, it would be possible for this adversary to compromise the confidentiality in case of encryption, or integrity and authenticity in case of digital signatures, of the information exchanged between these two users.

Currently, these systems can be implemented in a number of ways that provide different levels of assurance, against different threat models. The most popular implementations, e.g. SKS Keyserver [1], or Internet X.509 Public Key Infrastructure [2], are based on centralized certification authorities. They provide good availability and ease the key management operations (e.g. revocation or update). Yet, they all trust on the PKI provider to maintain the integrity of the keys and its binding to users identities, and moreover place this provider in a privileged position to conduct surveillance of users interactions. Recent proposals attempt to provide consistency via transparency logs, maintained and exchanged by independent PKI providers, and implemented as hash chains. Systems such as CONIKS [3] make *equivocations* by an email provider about users' public key material detectable and accountable.

Extending those ideas to a decentralized setting introduces significant challenges. By definition, in decentralized scenarios there is no actor that is globally trusted by all nodes in the system. Thus, the high-integrity and authenticity properties that usually rely on statements made by such trusted entity (e.g., identity-key bindings signed by a CONIKS provider), must be kept *without reference* to these trusted providers. Instead, all claims and structures need to be checked with reference to information *within the system*, i.e., held by the nodes themselves, considerably increasing the system's complexity. Additionally, the fact that nodes are called to make claims about others raises serious concerns about privacy. For instance, if nodes are users, those claims leak their contacts, and potentially their communication patterns and other interactions.

In this work we present ClaimChain, a PKI design that aims to tackle the problems mentioned above. The following are the objectives that we aimed to achieve in ClaimChain design and implementation.

Decentralized operation. To avoid the issue of centralization of trust inherent to traditional PKI designs, ClaimChain should be able to function in decentralized setting, with no reliance on trusted entities.

To this extent, ClaimChain relies on hash chains, that we call ClaimChains, and on *cross references across chains* to ensure the integrity of cryptographic key material and to support the provision of evidence about the binding between keys and identities. These hash chains represent the state of belief of one user about her own keys, and other users' ClaimChains. Each block in the chain contains all information necessary to access such state.

Flexibility, scalability, and ease of deployment. Even though we require that fully decentralized mode of operation is supported, ClaimChain should be flexible in terms of the context it is deployed. At one extreme, ClaimChains may reside locally on nodes, which are responsible of providing access to their claims. One way to enable such access is to embed ClaimChain within messaging systems, be it traditional email or modern chat applications, using the messages themselves as vehicle to transport information about bindings. On the other extreme, an online service that provides users with an API to store and access claims may be used. Moreover, ClaimChain should support hybrid modes of operation, in which different users may use different deployment contexts, still being able to inter-operate.

Regardless of the operation mode, the core security properties of ClaimChain must not rely on the fact that infrastructure in which claims reside is trusted for being honest. Instead, security and privacy must be guaranteed by cryptographic mechanisms.

Futhermore, ClaimChain should not have significant impact on computation time and bandwidth size of the users. Finally, as in any decentralized system, ClaimChain must assume that users may not be online at all times, cannot have a synchronized view of all state in the system, or even time; and for scalability has to operate even if users have a small fraction of information about others.

Privacy and non-equivocation. ClaimChain must provide the means to control who can access users' claims to guarantee privacy. We facilitate this through the use of cryptographic access control: using cryptographic access tokens, each user of the system indicates which other users are allowed read access to the claims they make. To all the non-authorized users, we ensure unlinkability across blocks through the use of nonces to randomize the claims' encryptions, such that it is not possible to distinguish how users' beliefs change over time.

As a side effect, privacy further complicates the notion of non-equivocation, since by design different users may have different views of other users' claims. This means that users may provide different versions of beliefs to different readers, which may disrupt efficient social resolution of key-identity bindings. ClaimChain, therefore, should enforce that the users can not equivocate. We provide a formal definition of non-equivocation, and outline suitable mechanisms that allow to reliably and efficiently prevent it. This is done without sacrificing privacy, i.e., minimizing the leakage of user's friendship networks – a problem that plagues decentralized PKI systems such as the PGP 'web-of-trust' [4].

The rest of this document is organized as follows. We review related work in Section 2. Section 3 presents the design of ClaimChain, describing its building blocks and providing a detailed specification of its operation, and explaining how it can be deployed to implement a decentralized PKI. The security and privacy properties of ClaimChain are described and proved in Section 5, and we evaluate the performance of ClaimChain in terms of computation, storage, and capability to propagate key bindings in Section 5.

2 Related work

This section provides a review of deployed PKI systems, that are the predominant centralized instantiations of Identity Systems, with a focus on those designs that aim in vouching for the authenticity of a binding between an email address and a PGP public key.

Trusted PKI Infrastructure. The default PKI most mail client agents use for submitting and retrieving PGP public keys is the SKS Keyserver [1]. Anyone can upload a binding to an SKS keyserver, which may end up holding many bindings for the same email to different public keys. Therefore there are no guarantees about which of the bindings, if any at all, is authentic.

The next generation of PGP PKI systems [5, 6] introduced an access control mechanism based on proving ownership of the email address involved in the binding by means of a confirmation link send to that address. Bindings get published only after they have been confirmed and only one binding per email address is kept. This model does not protect users from malicious key servers who advertise fake bindings, malicious email providers who update the binding of a user without his consent, or adversaries who could intercept the verification emails via network traffic.

The Nyms Identity Directory [7] complements email ownership verification with the use of "Trusted Notaries", which also sign bindings, and a "Network Perspective auditing mechanism", which cross-checks the bindings across providers to prevent them from equivocating about users' bindings. This approach has been taken up by Nicknym [8] to collect evidence about authenticity of keys; creating a Federated Web of Trust in which users are able to resolve the authenticity of a binding by following a friend-of-a-friend chain of signatures of notaries in an automatic way.

All the above systems can be complemented by CONIKS [3], which can be seen as a modification to Certificate Transparency (CT) [9] applied to PGP PKI systems. CONIKS equips providers with the means to create an auditable log of their users' public keys. Such log ensures the consistency throughout time of a binding of a user email address to the respective public key material. Effectively, equivocations about a user's public key by their provider are easily detectable and accountable.

The technological building blocks of CONIKS are essentially the same as the ones we use to build ClaimChain: we use verifiable random functions (VRF) and Merke trees to ensure high-integrity, and non-equivocation. However, as mentioned in the introduction, the challenges derived from moving to a decentralized scenario directly affect the way in which such primitives are combined to provide the same security properties.

Other key management solutions focus on facilitating key validation across devices of a user. Kokoris-Kogias et al. [10] propose a system in which users rely on a publicly reachable set of servers that form a cothority [11] (i.e., a decentralized set of authorities that collectively sign statements) that provides blockchain management as a service. Users manage their keys using this blockchain to publish their status, i.e., update, add, or remove keys and bindings. In turn, this blockchain can be accessed by others to validate bindings.

An example of a functioning decentralized PKI is Blockstack [12]. Blockstack uses a global system-wide Namecoin blockchain as a high-integrity store, with Bitcoin proof-of-work consensus mechanism [13] used by nodes to agree on the latest state of the system. Such approach is certainly feasible as demonstrated by widespread adoption of Bitcoin and Bitcoin-like cryptocurrencies, yet it comes with a number of drawbacks. Since the users need to agree on the state of the whole system at each point in time, the latency, storage and bandwidth cost increase significantly with the number of users. In ClaimChain we opt out from using a shared global state that has to be agreed upon by majority of nodes, because of these additional computational and bandwidth costs. Instead, we chose to have all users maintaining own hash chains corresponding

to their own claims and their local views of other users' states. In such setting, local consensuses about the partial state of the system arise within cliques of communicating users, rather than a single global consensus about the state of all participants. This allows for faster propagation of updates, and ensures that only relevant portion of the state of the ClaimChain nodes is being considered and is transmitted by users.

Social validation. An alternative to having a trusted entity, or entities, running the PKI is to rely on a social trust establishment mechanism to validate the authenticity of cryptographic keys. The paradigmatic example of such a mechanisms is the PGP Web of Trust (WoT) [4]. Conceptually, the WoT is similar to the endorsements by Trusted Notaries in Nyms [7], but in this case *any* PGP key owner can vouch for the validity of a binding. To decide whether a binding is authentic users consider vouches of their friends, and recursively those of the friends of their friends.

While the WoT is decentralized by nature, and thus does not have the disadvantages of centralized systems, it proved to be unusable in practice. A main reason for such failure are the difficulties associated to keys' management and validation. The former refers to the difficulty to manage the lifecycle of keys (generation, revocation, distribution) without a centralized server such as those mentioned above. The latter refers to the difficulty of authenticating the whole WoT, which facilitates equivocation attacks. Both problems are aggravated by the lack of automatism behind the mechanisms used by the WoT that require many user interactions, and make it complicated to trace back actions to inform decisions. Moreover, the current implementation of the WoT raises serious privacy concerns, since the use of vouching leaks the social relationships among users.

An alternative to alleviate the problems above is that taken by Keybase [14]. First, to hinder equivocation attacks based on taking over the social network, Keybase users publish statements about their binding in various social media or website domains they control that others can use as trust roots when validating the authenticity of the advertised keys. The key insight is that it is difficult for an adversary to gain access to existing social media accounts, and create genuine-looking fake accounts is expensive. Second, Keybase signature mechanisms ensure that signatures are not only public but ordered, i.e., cannot be rolled back, facilitating the identification of a problem. In addition, the Keybase provider holds a Merkle tree structure that, similarly to CONIKS, prevents the server from performing equivocation attacks. To further hinder the attack the root of this tree is published as a Bitcoin transaction.

3 ClaimChain design

This section introduces the design of the ClaimChain infrastructure. First we outline the goals of the system, to then delve into the details of its building blocks implementation.

3.1 Preliminaries

3.1.1 Cryptographic preliminaries

Our system relies on properties of *hash functions* to reduce integrity checks of a large, growing, set of statements about names and keys (claims), to a simple integrity check of a short fixed size sting of bits – the *head* of a *hash chain* or a *Merkle tree*. The ability to summarize all information known to a user about both her own and others' statements in such a short form, enables users inexpensive sharing and replication, while achieving high degrees of non-equivocation, and allows for easy replication and federation of the identity information. Further, cryptographic

primitives, such as *digital signatures* allow for self-authenticated updates, and *encryption*, and *verifiable random functions* allow us to implement some privacy features at the same time ensuring non-equivocation.

We use the following cryptographic primitives:

- Symmetric authenticated encryption scheme $\Pi_E = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. We assume Π_E is IND-CCA secure.
- Signature scheme $\Pi_S = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$, secure under selective forgery attacks.
- Two-party non-interactive key exchange protocol $\Pi_{KE} = (\mathsf{Gen}, \mathsf{SharedKey})$.
- Hash function scheme $\Pi_H = (\mathsf{Gen}_H, H)$, resistant to second-preimage attacks. Throughout this section for simplicity of notation H implicitly means H^s, where $s \leftarrow \mathsf{Gen}_H(1^\lambda)$ for some security parameter λ, when λ is clear from the context.
- Verifiable random function scheme $\Pi_{\mathrm{VRF}} = (\mathsf{Gen}, \mathsf{VRF}, \mathsf{Vrfy})$.
- Cryptographically secure pseudo-random number generator PRNG.

We denote string concatenation as $\|$.

3.2 ClaimChain outline

ClaimChain allows a user to create a 'ClaimChain', which is a repository of claims the user makes about herself, or other users, maintained over time. Each user may have one or multiple such ClaimChains, for example associated to multiple devices or multiple pseudonyms.

A ClaimChain needs to be an accurate witness of a user's belief into their own information, as well as claims about others, as those beliefs evolve over time. To achieve this ClaimChain implements ClaimChains as cryptographic chains of *blocks* of claims. Each block of the chain contains enough information to authenticate past blocks as being part of the chain, as well as validate future blocks as being valid updates. Thus, a user with access to a block of a chain that they believe is authoritative, i.e., considered to provide correct information, may audit past states of the chain – and past claims or user meta-data – and may also authenticate the validity of newer blocks. This ensures that users trusting a chain can efficiently accept the latest state of its owner's claims.

A user stores three types of information in ClaimChain:

- **Own metadata.** A ClaimChain contains up to date information about its owner. This information may include the owner identity, such as their screen name, real name, email or chat identifiers; as well as authentication material for their cryptographic public keys, i.e., verification keys to support digital signatures, or encryption keys to support confidential messaging. Claims about a user's own metadata are initially self asserted, and gain credibility for other users through being certified by other users.
- **Claims about other users.** A user may include claims about others into their ClaimChain. The most simple claim is endorsing another ClaimChain as being authoritative – which implies that its metadata binding identifiers to keys are correct in the user's view. Other claims may also be included, such as statements about the latest state of others' ClaimChains (which we call cross referencing), links to other ClaimChains by the same user, or revocation of ClaimChains that are known to have been compromised.

- **System integrity information.** A ClaimChain needs to contain enough information to authenticate all past states of the ClaimChain repository, as well as to authenticate valid future updates. For this purpose, it contains keying material and commitments that facilitate the authentication. In order to enable efficient operations without the need for another party to have full visibility of all claims in the chain, we augment ClaimChains with information to facilitate those operations, namely quick cryptographic links to past states, as well as roots of high-integrity data structures such as Merkle trees.

Each block of a ClaimChain includes all meta-data, and all claims that its owner endorses at the point in time the block is generated. We deliberately chose to replicate the full database of claims and full state of the owner of the chain, to keep the design of access control and non-repudiation simple. Alternative designs, relying on only including updates in each block proved elusive optimizations, and might be a good avenue for future work. However, we opt for replicating all user state into each block of her ClaimChain, and provide efficient representation, including partial representations of this complete state instead, to deal with efficiency.

In a nutshell, we envision that users will use ClaimChain to create one ClaimChain for each of their own identities for which they want to make claims about. When a user wants to establish the validity of a claim, she uses other users' ClaimChains to gather evidence about others' beliefs on this chain. The evidence is processed according to some policy defined by the user performing the validation, to decide upon the validity of the chain. We stress that the goal of ClaimChain is not to decide on the best validation policy, but to ensure that users can gather evidence in a decentralized and secure manner by ensuring the following security properties, which are defined formally in Section 5:

- **Authenticity** of the information stored in a ClaimChain: the information in a ClaimChain has been input by the owner of the chain.
- **Integrity** of the information stored in a ClaimChain: this information has not been modified since it was added to the chain.
- **Privacy** of the information stored in a ClaimChain: for every claim in every block, if readers are honest and do not distribute any secret information, only readers authorized by the chain owner can access the content of the claim.
- **Non-equivocation** about the information stored in a ClaimChain: for every claim in every block, regardless of the access control policy implemented, an owner of a chain cannot provide two different contents about the claim to different readers.

ClaimChains security and privacy properties are ensured by the cryptographic mechanisms used to build it. Thus, the security of ClaimChain is independent of where the ClaimChains reside. For instance, they can be stored locally by users and transmitted to others when they need access to validate claims; or they can be stored at a central server where others can access them. Note that in the second case, cryptographic access control mechanisms prevent the server from learning information stored in the ClaimChain. We note that the storage provider may in this case learn information analysing the acccess pattern to users chains. See 3.3.5 for more details about this issue.

3.3 ClaimChain building blocks

The core building element in ClaimChain is a *block*: a data structure that consists of some *payload*, along with a public-key digital signature on that payload. The payload contains information that can be used by authorized readers to learn the owner's statements, that we call

claims. Independent blocks are full snapshots of the owner's *state* at a given time. A block serves as a self-contained commitment to this state, reflecting the owners' belief about her own and others' claims.

For auditability reasons we link chronological sequences of blocks into *ClaimChains*. Chronology is implemented as a hash chain, with each block including a reference to past blocks according to deterministic skip list rules [15]. Given the latest block of a ClaimChain, also referred to as the *head* of the ClaimChain, one can traverse the whole history back to the first block at the origin of the ClaimChain, called *genesis block*. Thanks to these references to previous blocks, a block is not only a commitment to the current state, but also serves as a commitment to the full history of past states.

The design of ClaimChain in principle could handle any generic claims about user beliefs. However, for the sake of simplicity, in this document we define a restricted set of claims that are related to PKI systems. In this setting, ClaimChain claims may be statements about key material owned by the owner of the ClaimChain, key material owned by another person with label, cross-references to other ClaimChains (i.e., claims about the state of other ClaimChains), and statements about the metadata of the ClaimChain for maintainance.

ClaimChain supports the existence of *private claims*, that only certain readers can access. Such privacy could enable an owner to present different views of a given claim to different people. However, we ensure non-equivocation, i.e., that this attack is not possible using cryptographic controls (see Section 3.3.2).

As mentioned before, objects in a ClaimChain block may be either stored in the block payload directly, or may be referenced using their cryptographic hash representation. Readers should be able to retrieve these objects from the same storage medium as the block; or they should be provided with those structures as part of an interactive protocol leveraging ClaimChain for integrity. We note that, in the former case, storage providers need not to be trusted for not tampering with user data. This is because malicious storage providers cannot equivocate about referenced objects, and integrity is guaranteed through the cryptographic hash imprint of the object in the signed block payload.

In the following sections, we give a more detailed technical description of the ClaimChain's building blocks at both high level and low level.

3.3.1 User meta-data in the ClaimChain

Each ClaimChain block, including the genesis block, contains meta-data. This meta-data includes the cryptographic keys that can be used to authenticate the owner and encrypt data for her, the identity of the ClaimChain owner, and other information necessary to read claims within this block. More specifically, the following information is included, in clear for all to see:

- **Identities.** These are owner identities whose cryptographic associated information are stored in the ClaimChain. For instance, the email address, twitter handle, signal number, POTS number, web page, of the owner of the chain. These are used to extract labels that characterize this chain, and guide other users to select an appropriate ClaimChain to use to obtain cryptographic keys when trying to communicate with a partner.

  The metadata can also include attributes reflecting other devices or ClaimChains belonging to the same owner. These attributes are used by the ClaimChain owner to point others to where to find information about her alternative identities or devices.

  It is crucial to note that a change in the identity information should not be automatically accepted without re-validation. Otherwise, dishonest contacts could change their identities to impersonate others.

- **Cryptographic keys.** Two types of keys are included in the metadata: keys that are necessary for the operation of the ClaimChain, and keys that are useful for applications. The former include the current signing key of the owner, $\mathsf{pk}_{\mathsf{SIG}}$, that is used to authenticate new blocks of the ClaimChain; the current key to compute verifiable random functions, $\mathsf{pk}_{\mathsf{VRF}}$, used to support non-equivocation as explained below; and a revocation key, that is included to enable revocation in case the signing key is compromised. The latter consists of an Diffie-Hellman key for key derivation, which we call the ClaimChain encryption key, $\mathsf{pk}_{\mathsf{DH}}$. This key is used, in combination with other ClaimChains encryption keys, to implement private claims.

- **Storage information.** Optionally the owner of the ClaimChain can provide hints about services where the data, and latest heads, of this ClaimChain can be stored and retrieved. This supports an on-line operation of ClaimChain, or a full audit of the history of a ClaimChain. For asynchronous settings, information on the email or other endpoint where data about this ClaimChain may be requested, is included.

3.3.2 Claims and the block claim map

Claims within a block are the assertive statements that carry information the owner of the chain wants to share at that particular point in time. Each claim is indexed by a 'label' that is a well-known identifier associated with the person, identity, network end point, or whatever other identifier the claim refers to. As an example, a claim may be labelled as 'alice@gmail.com' if it refers to a belief the ClaimChain owner has about Alice's gmail account's key or associated ClaimChain.

All claims in a ClaimChain block are encrypted and indexed using a verifiable random function (VRF), under the correspondent public key $\mathsf{pk}_{\mathsf{VRF}}$ published in the ClaimChain metadata, and 'salted' using a nonce published in the same block. At a high-level, consider a claim $\mathsf{claim\_body}$ with label $\mathsf{claim\_label}$, that is to be included in a ClaimChain block with nonce, under the ClaimChain $\mathsf{pk}_{\mathsf{VRF}}$. We first derive a key for this claim, by generating a value k using a verifiable random function as $k, \mathsf{proof} = \mathsf{VRF}_{\mathsf{pk}_{\mathsf{VRF}}}(\mathsf{claim\_label} \parallel \mathsf{nonce})$, where proof is needed to be able to verify the VRF value. The index of the claim is derived using this shared secret as $L = H_1(K)$ and an encryption key is derived as $K_E = H_2(K)$ where H_1 and H_2 are cryptographic hash functions. The body of the claim and proof are then jointly encrypted as: $C = \mathsf{Enc}_{K_E}(\mathsf{proof} \parallel \mathsf{claim\_body})$. The tuple (L, C) represents the encrypted claim.

A ClaimChain block includes the list of all tuples (L, C) of claims encrypted under the block's nonce. We consider the L component to be a 'key' of the claim in that block (in the sense of a key-value store, not a cryptographic key, hence we also call it a 'lookup key'), and the C component to be the 'value' of the encrypted claim. The tuples are stored in a data structure that provides a 'map' interface storing the values of the encrypted claims, indexed by their keys in the block – which we call the *claim map*. As we discuss later, we instantiate this structure as a sorted Merkle tree adapted to provide non-equivocation.

Given a VRF value k for a $\mathsf{claim\_label}$, a tuple (L, C), the block nonce and the public key $\mathsf{pk}_{\mathsf{VRF}}$ associated with the block, anyone may verify that (L, C) is valid for a $\mathsf{claim\_label}$. First, we derive the claim key L using k and H_1. The result should be equal to the given L. Second, the decryption key K_E may be derived, and used to decrypt C to recover the proof and $\mathsf{claim\_body}$. Finally, the proof may be used in conjunction with the public key $\mathsf{pk}_{\mathsf{VRF}}$, the $\mathsf{claim\_label}$ to verify the validity of the given VRF value k. Users must always check that a tuple (L, C) is valid for a claim label, before using the decrypted claim.

This scheme for encoding and decoding of claims offers a number two distinct security advantages:

1. A valid claim only has a unique VRF value k for a given block nonce, and thus a unique lookup key L. As a result this key may be used to support non-equivocation for claims relating to this label, as we discuss below.

2. Without knowledge of the VRF value k, the tuple (L, C) leaks no information about the claim label or the claim body – preserving privacy. This allows us to preserve the privacy of claims, and the privacy of the social graph of who makes claims about whom. We use cryptographic access control, described below, that provides selective access to these private claims by sharing with some users, and not others, the appropriate VRF values.

We note that owners may want to also share "public" claims, i.e., claims that are readable by anybody with the access to the block. These public claims may be supported in a variety of ways: (1) They could be stored directly as a tuple (claim_label, claim_body) in the claim map; (2) instead of using a VRF to derive k we could use a traditional hash function; (3) the VRF value could be derived using a public key with a well known private key, instead of a secret one; (4) the VRF values could be included in clear elsewhere in the ClaimChain. To keep the complexity of the code down, we opt for the last solution, since it allows us to implement a single encoding and decoding mechanism that uniformly supports private and public claims. We discuss further how public claims are handled, and how their VRF values become known, in the Section 3.3.3.

3.3.3 Access control, public access and the capabilities map

Recall that the key goal of ClaimChain is to support a decentralized PKI, with the additional requirement that users' claims about others must not leak their social graph in an uncontrolled manner, i.e., an adversary must not be able to learn who is making claims about whom. To achieve this, we apply cryptographic access control to restrict reading access to claims. Additionally, this prevents non-authorized users to infer that a claim for a specific label is present within a block.

Abstractly, we consider an access control matrix [16] where subjects represent potential readers of claims; objects are labels of claims; and the only access right is 'read'. We assume that the owner of a ClaimChain, through an appropriate user interface or other mechanism, specifies this access control matrix. The objective of our system is then to ensure that only reads allowed by the policy represented by this matrix can be performed – i.e. only users with a 'read' privilege should be allowed to detect that a claim for a label is present in the claim map of a block, and decrypt this claim. We implement this using cryptographic controls to regulate access to the VRF values associated to each label.

We assume that for each subject with rights in the access control matrix, the owner of the ClaimChain has access to their up-to-date Diffie-Hellman public key (i.e., the current encryption key in ClaimChains of these users). We also assume that the subject, at the time of reading, will have access to a Diffie-Hellman public key associated with the ClaimChain containing the claims to be read – i.e., the encryption key included in the metadata of the block where the claim resides. We use those keys to derive pairwise encryption keys between the owner of the ClaimChain and reading subjects, under which we encrypt the VRF values of the labels these readers have access to in the access control matrix.

The access control matrix is stored in a map structure within the block that we call the *capabilities map*. There are two options for the granularity at which the access control matrix may be encoded:

- **All-labels Capability.** The first option is to store the full list of encrypted VRFs for labels that a reader has access to under a look-up key associated to the reader label. This means that a reader finds all claims to which she has access at once, independently of her knowing their labels or having interest in accessing them. The DH secret key of the ClaimChain $\mathsf{sk}_{\mathsf{DH}}$ and the public of the subject Alice $\mathsf{pk}^A_{\mathsf{DH}}$, are used to derive a shared secret $s \leftarrow \mathsf{SharedSecret}(\mathsf{sk}_{\mathsf{DH}}, \mathsf{pk}^A_{\mathsf{DH}})$. Then, using the block nonce, a capability entry lookup key L^A and an encryption key K^A_E are derived as:

$$\begin{aligned} L^A &= H_3(\mathsf{nonce} \parallel s) \\ K^A_E &= H_4(\mathsf{nonce} \parallel s), \end{aligned}$$

where H_3, H_4 are cryptographic hash functions. Finally, all VRF values associated with the labels of claims to which Alice has access are encrypted:

$$P^A = \mathsf{Enc}_{K^A_E}([\ldots k_i \ldots]). \tag{1}$$

We store a tuple (L^A, P^A) for every reader in the *capabilities map*.

- **Single-label Capability.** Alternatively, each access right may be encrypted separately under a look-up key and cryptographic key specific to the reader and the label to be read. In this case, a reader can only find claims for which she knows their label. The full set of reading rights for a given subject is given by a number of these single-label capabilities. Concretely, for each subject-label pair that has a 'read' access right we derive:

$$\begin{aligned} L^{A,\ell} &= H_3(\mathsf{nonce} \parallel s \parallel \mathsf{claim\_label}) \\ K^{A,\ell}_E &= H_4(\mathsf{nonce} \parallel s \parallel \mathsf{claim\_label}). \end{aligned} \tag{2}$$

Then we encrypt the VRF value for the specific label using $K^{A,\ell}_E$ as:

$$P^{A,\ell} = \mathsf{Enc}_{K^{A,\ell}_E}(k_\ell). \tag{3}$$

The pairs $(L_{A,\ell}, P_{A,\ell})$ for all subject-label pairs are stored into the *capabilities map*.

A user wishing to access a claim within a ClaimChain uses the information in the *capabilities map* to retrieve the VRF value for the label of interest, within the block. This VRF value is then used to derive the look-up keys and decryption keys necessary to look-up and decrypt the capability within the *claims map*. This gives access to the full list of VRF values, in case of a 'full capability' scheme, or to a specific label in the case of the 'single-label capability' scheme. We note that the VRF value cannot be verified before the claim has been retrieved and verified, since the proof necessary is not included in the capability map.

Public claims. Besides private claims, ClaimChain also supports public claims that can be accessed by anyone. Public claims and private claims have the same representation in the claims map, namely they are indexed and encrypted using keys derived from a VRF that is different in every block. However, in the case of public claims the VRF corresponding to the label of the public claim is provided 'in the clear' within the capabilities map, for anyone to lookup and retrieve.

Consider a public claim with a label claim_label and VRF_i. Within a block with nonce we derive the lookup key $L = H_3(\mathsf{nonce} \parallel \texttt{'public'} \parallel \mathsf{claim\_label})$, and define P to be a VRF value of the claim_label. The tuple (L, P) is the public capability corresponding to the public claim, and is included in the capabilities map. To retrieve and check a public claim a reader

would derive publicly the lookup key L, recover the VRF value, and then use it to derive all information necessary to look up the encrypted claim in the claims map.

We note that the homogeneity between public and private claims, in terms of their representation in the claims' map, is necessary to avoid equivocation. The owner of a ClaimChain can only create a single representation of the VRF value per block per claim label. Thus even if they include a mixture of capabilities for public and private claims for the label in the capabilities map, those referring to the same label will always resolve to the same VRF value.

3.3.4 Block & chain structure

Each block $B_i = (X_i, \sigma_i)$ in our chain comprises a payload X_i, and a signature $\sigma_i = \mathsf{Sign}_{\mathsf{sk}^t_{\mathsf{SIG}}}(H(X_i))$ for a signing key pair $(\mathsf{pk}^t_{\mathsf{SIG}}, \mathsf{sk}^t_{\mathsf{SIG}})$:

$$B_0 = \left(X_0,\ \sigma_0 = \mathsf{Sign}_{\mathsf{sk}^0_{\mathsf{SIG}}}(H(X_0))\right) \qquad \text{(genesis block)}$$

$$B_i = \left(X_i,\ \sigma_i = \mathsf{Sign}_{\mathsf{sk}^{i-1}_{\mathsf{SIG}}}(H(X_i))\right), i > 0 \qquad \text{(regular block)}$$

We note that *the signature key for every block varies.* The inclusion of a signature per-block allows the authentication of subsequent blocks. To be authenticated a B_i must have a valid signature under the verification key indicated in the payload of block B_{i-1}. The genesis block of the ClaimChain is 'self-signed' with a key pair designated in the initial payload. The selection, validity and revocation of signing keys will be discussed later in this report.
The prototype of a Block payload X_i has the following fields:

- **Version.** A version number associated with the code to interpret this ClaimChain.
- **Block index.** The sequential number of this block, i.e., its position in the chain. The index of the genesis block is 0.
- **Timestamp.** A timestamp representing when the blcok was generated. The timestamp is a unix epoch, at a granularity pre-defined by the version.
- **Nonce.** A fresh cryptographic nonce that is used to 'salt' all cryptographic operations within the block. This nonce ensures that information across blocks is not linkable.
- **Meta-data.** The identities and keys associated with this block. These meta-data are potentially updated at every block.
- **Non-equivocable map.** The root of a Merkle tree (see below) mapping keys to values in a non-equivocable fashion. This map contains both the claims map and the capabilities map.[1] It has two key properties: i) a key can only be resolved to a single value, and ii) given the root it is easy to produce a proof for the correct value (or a proof of non-inclusion of any value).
- **Pointers to previous blocks.** A hash of the previous block, as well as hashes of past blocks. These hashes, that form ClaimChain, generate a high-integrity skip list (see below) to allow faster authentication of past ClaimChain states.

[1] Even though the mapping of capability lookup keys to encrypted capability entries, and the mapping of claim lookup keys to encrypted claims are semantically different, we choose to merge these two maps into one structure, mainly for simplicity.

Non-equivocable Merkle tree. Within each block we include the root of a Merkle Tree representing a non-equivocable high-integrity key-value map. This map is used to store the claim map and the capabilities map for the block. It enables the generation, or verification, of efficient proofs of inclusion or exclusion of specific claims or capabilities.

The Merkle tree is a data structure that is composed of two types of nodes:

$$\begin{aligned} &\mathsf{Node}: \\ \mathsf{Internal} &= (\mathsf{pivot},\ \mathsf{left} : H(\mathsf{Node}),\ \mathsf{right} : H(\mathsf{Node})) \\ \mathsf{Leaf} &= (\mathsf{key}, \mathsf{value}) \end{aligned}$$

Each Internal node contains a pivot key, and the invariant of the structure is that any Leaf nodes in the left sub-tree will have keys smaller than the pivot, and any Leaf nodes to the right have keys equal or larger than the pivot. Internal nodes store the hash of the node representing the left and right sub-tree. This effectively creates a Merkle tree in which the hash of the root node is a succinct authenticator committing to the full sub-tree (subject to the security of the hash function).

A proof of inclusion of a key-value pair in the tree involves disclosing the full resolution path of nodes from the root of the tree to the sought leaf. Similarly, a proof of non-inclusion involves disclosing the failing resolution path from the root of the tree to the leaf nodes that are not the sought key-value. We note that for a single key only one value is to be stored. Any violation of this invariant may be detected when the proof of inclusion or exclusion are checked – thus the creator of the tree does not need to be trusted to enforce this invariant.

High-integrity skip list. Traditional hashchains and blockchains only contain the hash of the previous block, requiring linear verification time to authenticate blocks in the past. Blocks in a ClaimChain include hashes of blocks beyond the previous one to enable verification to be faster than linear. By including a selection of hashes of past blocks (including the one immediately preceding the block) the cost is reduced to $\mathcal{O}(\log(i-j))$ where i is the index of the latest block, and j is the index of the past block to be verified as belonging to the ClaimChain.

Concretely, a block includes some subset of hashes F_i to previous blocks $F_i = \{(j, H(B_j)) | j \in J(i)\}$ for a set of indices $J(i)$, such that $\forall j \in J(i) : j < i$. The indexes $J(i)$ are chosen to mirror the structure of a deterministic skip-list:

$$J(i) \equiv \{\forall t \in \mathbb{Z}^*. \quad i - 1 - ((i-1) \mod (2^t))\}.$$

Notionally the indexes $J(i)$ fall on the 'tick marks' of a binary ruler, one per height of tick-mark preceding block index i. For example $J(127) \equiv \{64, 96, 0, 112, 120, 124, 126\}$ and $J(128) \equiv \{96, 64, 0,\ 112, 120, 124, 126, 127\}$.

These sequences $J(i)$ have valuable properties that allow for efficient authentication of past ClaimChain blocks efficiently: The sequence for $J(i)$ ensures that given an index $j < i$, it contains a a number that is at least $(i-j)/2$ closer to j than i. This ensures that when authenticating block j departing from block i, it is possible to retrieve a past block that is at least half the distance between blocks i and j. This block will also contain past hashes that can be used to recursively authenticate any past block in $\mathcal{O}(\log(i-j))$ jumps.

The size of the sequence $J(i)$ is of length $O(\log i)$ and therefore occupies little space in the block, and grows slowly as the chain grows. Furthermore, consecutive $J(i)$ and $J(i+1)$ share a large number of common elements, meaning that fewer hashes need to be included in every block to ensure that fast resolution can be performed (but we do not explore this pruning further).

3.3.5 Object store abstraction

To support the operation of ClaimChain we assume the existence of a key-value store, whose keys are the hash of the values, in which all objects relating to ClaimChains (blocks, tree nodes, encrypted blobs) reside. This storage infrastructure provides the following interface to a ClaimChain client:

Get(store, h)	Returns the object with hash $H(\text{object}) = h$ from `store`.
Put(store, `object`)	Saves object to the store.

We call this abstraction an Object Store. It has a number of key properties:

- **Self-certification.** Given an Object Store it is easy to verify its integrity, by checking the invariant that all keys are the hashes of the respective objects they map to. However, its completeness cannot be guaranteed in general.
- **Conflict-free merge.** Given two valid Object Stores it is easy to merge them, by simply constructing a store with the union of their key-value pairs. This operation can be performed recursively to merge multiple stores. Note that merges cannot lead to conflicts or inconsistencies.
- **Tolerance to partial views.** ClaimChain's use of the Object Store guarantees that incomplete stores may result in failed attempts to authenticate data structures, or failure to check inclusion or non-inclusion of claims. However, an incomplete store cannot lead to an erroneous inference on the authenticity, inclusion or exclusion or any chain or claim.
- **Flexible distribution.** Object Stores may be replicated across on-line infrastructure, and mirrored locally on ClaimChain's clients. Due to the properties of the Object Store it may be implemented in a sharded manner on-line to increase performance, and partial off-line operation of partial stores cannot lead to errors.

The above properties are crucial in supporting flexible on-line, off-line, centralized or decentralized operations for ClaimChain. However, we note that the Object Store is an abstraction, and concrete designs must opt for a specific instantiation of this abstraction. In particular, we describe below how such an Object Store can be implemented in a on-line setting (using a collection of internet services), as well as asynchronous decentralized setting (as necessary to support entirely decentralized email integration).

On-line service instantiation. The object store abstraction can be implemented on top of a publicly accessible on-line key-value store even without authorization or authentication. Even in such case, the mechanisms in ClaimChain still ensure all the desired security properties (see Section 3.2), other than query privacy and resistance to denial of service attacks, are fulfilled . Clients may rely on one, or multiple on-line services to further improve availability, but for denial of service to be solved, at least one *honest* provider has to be highly available. In all cases clients must check the integrity of returned key-value pairs by checking that the lookup keys are hash values of a corresponding object.

Regarding query privacy, basic guarantees against a network adversary can be provided through standard padded encrypted channels between the on-line services and the clients. More sophisticated protection mechanisms are needed to improve query privacy against an *honest but curious* storage provider. Specifically, if the accesses are authenticated, a (ε-)PIR mechanism [17] needs to be used to lookup and return key-values.

Off-line asynchronous instantiation. A provider-less, off-line instantiation is also possible. In this scenario each user maintains a local key-value store, that can be updated through gossip with other users, or by receiving out-of-band evidence about other user's ClaimChain status.

The store self-certification and conflict-free merge properties give flexibility to this design option, since it permits that evidence that does not come from an authoritative source to be included in the key-value store. It must be noted, however, that in this off-line setting it is not possible to guarantee that the Object Stores is complete. Yet, the properties of ClaimChain ensure that at worst some claim resolutions may fail, but would never return a false answer.

3.4 ClaimChain specification

This section describes in detail the operations that users need to perform to build and use a ClaimChain.

3.4.1 Basic operations

First we detail the basic operations required to build a ClaimChain as outlined in Sections 3.3.2 and 3.3.3. Concretely, we describe the encoding and decoding of capability entries and claims, as well as the computation of capability lookup keys.

Claim encoding. Before committing a claim with label claim_label and content claim_body to the ClaimChain, the owner must encode it. This process, shown in Algorithm 1, takes as input the label and content, as well as the current nonce and the secret key of the owner. Given this information the owner computes the VRF of the nonce and label, k and its associated proof. Subsequently, she obtains the lookup key L and encryption key. The latter is used to encrypt the claim. The function returns the encoded claim (L, C) itself, and the VRF value k.

Algorithm 1 Claim encoding

function EncodeClaim($\mathsf{sk}_{\mathsf{VRF}}$, nonce, claim_label, claim_body)
  $k, \mathsf{proof} \leftarrow \mathsf{VRF}_{\mathsf{sk}_{\mathsf{VRF}}}(\mathsf{nonce} \parallel \mathsf{claim\_label})$
  $L \leftarrow H_1(k)$
  $K_E \leftarrow H_2(k)$
  $C \leftarrow \mathsf{Enc}_{K_E}(\mathsf{proof} \parallel \mathsf{claim\_body})$
return $k, (L, C)$

Encoding of capabilities Given the VRF value k for a label computed using Algorithm 1 EncodeClaim, the owner can encode capability entries by encrypting k with keys shared with other users, which in turn are non-interactively derived from shared secrets. To compute a shared secret with each of the intended readers of the claim, the owner uses their public keys, denoted as $\mathsf{pk}^A_{\mathsf{DH}}$ for reader A. The procedure, described in Algorithm 2, first computes the shared secret which is used to encode per-user labels and compute per-user keys. These keys are used to encrypt the VRF value k required to read claims. The function returns the encoded capability for claim ℓ with claim_label to reader A.

Computing the capability lookup keys To be able to retrieve a capability entry for claim ℓ of interest with laclaim_label of interest from the map, a reader A first needs to compute this capability lookup key. This lookup key is derived from a shared secret, for which the reader needs to know the owner's $\mathsf{pk}^B_{\mathsf{DH}}$. The procedure, illustrated in Algorithm 3 computes this

Algorithm 2 Capability encoding

function EncodeCapability($\mathsf{sk}_{\mathsf{DH}}, \mathsf{pk}^A_{\mathsf{DH}}, \mathsf{nonce}, \mathsf{claim\_label}, k$)
  $s \leftarrow \mathsf{SharedSecret}(\mathsf{sk}_{\mathsf{DH}}, \mathsf{pk}^A_{\mathsf{DH}})$
  $L^{A,\ell} \leftarrow H_3(\mathsf{nonce} \parallel s \parallel \mathsf{claim\_label})$
  $K_E^{A,\ell} \leftarrow H_4(\mathsf{nonce} \parallel s \parallel \mathsf{claim\_label})$
  $P^{A,\ell} \leftarrow \mathsf{Enc}_{K_E^{A,\ell}}(k)$
return $(L^{A,\ell}, P^{A,\ell})$

shared secret and uses it to obtain the lookup key from the current nonce and the claim_label. The capability lookup key $L^{A,\ell}$ is returned to the reader.

Algorithm 3 Computing capability lookup key

function ComputeCapabilityLookupKey($\mathsf{sk}_{\mathsf{DH}}, \mathsf{pk}^B_{\mathsf{DH}}, \mathsf{nonce}, \mathsf{claim\_label}$)
  $s \leftarrow \mathsf{SharedSecret}(\mathsf{sk}_{\mathsf{DH}}, \mathsf{pk}^B_{\mathsf{DH}})$
  $L^{A,\ell} \leftarrow H_3(\mathsf{nonce} \parallel s \parallel \mathsf{claim\_label})$
return $L^{A,\ell}$

Decoding of capabilities Prior to reading a claim, a reader A needs to decode her capability, as described in Algorithm 4. To decode a capability entry $P^{A,\ell}$, A uses the ClaimChain owner's public key $\mathsf{pk}^B_{\mathsf{DH}}$ to obtain her shared secret, which in turns enables to derive the decryption key for the VRF value k and the encoded lookup key for the claim. The procedure returns the decrypted VRF value k, as well as the claim lookup key L for finding C in the claim map.

Algorithm 4 Capability decoding

function DecodeCapability($\mathsf{sk}_{\mathsf{DH}}, \mathsf{pk}^B_{\mathsf{DH}}, \mathsf{nonce}, \mathsf{claim\_label}, P^{A,\ell}$)
  $s \leftarrow \mathsf{SharedSecret}(\mathsf{sk}_{\mathsf{DH}}, \mathsf{pk}^B_{\mathsf{DH}})$
  $K_E^{A,\ell} \leftarrow H_4(\mathsf{nonce} \parallel s \parallel \mathsf{claim\_label})$
  $k \leftarrow \mathsf{Dec}_{K_E^{A,\ell}}(P^{A,\ell})$
  $L \leftarrow H_1(k)$
return k, L

Claim decoding. In order to read an encrypted claim C, a reader must decrypt it. Given the claim label, the current nonce, the VRF value k obtained from Algorithm 4, and the public key of the owner, the reader follows Algorithm 5. First she computes the encryption key for the claim, which she uses to recover the claim body and VRF proof. The proof must be verified and then the decrypted claim_body is returned.

3.4.2 Tree operations

ClaimChain requires a non-equivocable high-integrity key-value 'map' data structure, which we instantiate using a Merkle tree (Section 3.3.4). In this section we describe in detail how to construct and query Merkle trees to ensure the desired security properties.

Constructing the tree After encoding all the capabilities and claims using EncodeCapability and EncodeClaim procedures, the owner obtains a set of *map entries*. These are the key-value

Algorithm 5 Claim decoding

function DecodeClaim($\mathsf{pk}^B_{\mathsf{VRF}}, \mathsf{nonce}, \mathsf{claim\_label}, k, C$)
  $K_E \leftarrow H_2(k)$
  $\mathsf{proof} \parallel \mathsf{claim\_body} \leftarrow \mathsf{Dec}_{K_E}(C)$
  assert $\mathsf{Vrfy}_{\mathsf{pk}_{\mathsf{VRF}}}(K, \mathsf{proof}, \mathsf{nonce} \parallel \mathsf{claim\_label})$
return claim_body

pairs $S = \{..., (k, v), ...\}$ that the owner needs to put in the tree. This process, described in Algorithm 6, is at its core the construction of a binary search tree.

Note that we choose not to store the values themselves in tree leaves. In Algorithm 6 only hashes of values are stored in the leaves, while the values are put into an object store as content-addressable *blobs*. Once a reader obtains the leaf by querying the tree, she can retrieve a value itself by its hash from the store. The fact that the leaf nodes do not contain the claim bodies allows for efficient programmatic representation of trees. This is because all leaves, and all internal nodes, have the same size regardless of the nature of claims users wish to put in their ClaimChains, and thus the node representations can be aligned in memory.

Algorithm 6 Tree construction

function BuildTree(S, store)
  if S is $\{(k, v)\}$ **then** ▷ Base case
    leaf $\leftarrow$ Leaf$(k, H(v))$ ▷ Create a leaf containing the hash of the value
    Put(store, leaf)
    Put(store, v) ▷ Put the value itself into the store
    return H(leaf)
  else
    $\{(k^*, v^*), ...\} \leftarrow S$ ▷ Pick the pivot arbitrarily
    $(S^-, S^+) \leftarrow$ Partition(k^*, S) ▷ Partition the pairs w.r.t the pivot
    node $\leftarrow$ Node$(k^*,$ BuildTree$(S^-,$ store$),$ BuildTree$(S^+,$ store$))$
    Put(store, node)
    return H(node)

function Partition(k^*, S)
  $S^-, S^+ \leftarrow \{\ \}, \{\ \}$
  for (k, v) in S **do**
    if $k^* < k$ **then** ▷ Lexicographic comparison of binary strings
      $S^- \leftarrow S^- \cup \{(k, v)\}$ ▷ Pairs with keys less than k^*
    else
      $S^+ \leftarrow S^+ \cup \{(k, v)\}$ ▷ Pairs with keys greater or equal than k^*
  return (S^-, S^+)

Querying the tree Merkle trees in ClaimChain are used in a way that enables two distinct modes of operation, while having the same node representations. First, using a remote online object store, it should be possible to verifiably retrieve values from the tree only knowing the hash of the tree root and the entry lookup key. This can be done by retrieving the nodes from the store one by one starting from the known root. Second, a reader can obtain inclusion evidence of an entry as a result of running some interactive protocol with the owner or a "smart" storage

provider. Given this evidence, the hash of the tree root, and the lookup key, reader should be able to verify that the lookup key of interest is indeed included in the tree. These correspond to centralized or federated online deployment, and decentralized asynchronous mode respectively (see Section 4 below).

To capture both cases, we assume the reader has access to an object store containing a sufficient set of tree nodes and value blobs. In other words, the reader should be able to traverse the tree all the way from the root to the leaf that holds the value of interest, by retrieving objects from the store. In the decentralized mode of operation this implies that all the evidence obtained by the reader asynchronously has to be included in her local object store prior to checking whether some value is included or not. A reader essentially has to replicate a relevant portion of the owner's store.

The process of a reader querying the tree (QueryTree) is described in Algorithm 7. It involves traversing the tree starting from the root until the relevant leaf node is found, at each step using the pivot field to know whether to follow the left or right subtree.

Note that computation of evidence paths by owner in decentralized setting, and verification of evidence paths contained in the reader's local object store, is the same procedure. Thus, the ComputeEvidence procedure from Algorithm 7 is used by (a) any readers in ClaimChain as a subprocedure of QueryTree, and (b) by owners that need to compute evidence paths in decentralized setting.

Algorithm 7 Querying the tree and computing inclusion evidence

```
function QueryTree(MTR, k, store)
    evidence ← ComputeEvidence(MTR, k, store)
    [..., Leaf(k′, v′)] ← evidence
    assert k′ = k
    return v′

function ComputeEvidence(MTR, k, store)
    node ← Get(store, MTR)
    if node is Leaf then
        return node
    else if node = Node(pivot, left, right) then
        if k < pivot then
            evidence′ ← ComputeEvidence(left, k, store)
        else
            evidence′ ← ComputeEvidence(right, k, store)
        return Concat(node, evidence′)          ▷ Concat is list concatenation
```

3.4.3 High-level operations

We now describe in detail the high-level operations that users perform to make use of a ClaimChain. Concretely, the operations performed by the owner in order to update a ClaimChain with a given set of claims and an access control matrix, and the operations performed by the reader to retrieve a claim from a ClaimChain given its label. These internally use all of the basic encoding-decoding operations, and the tree operations from the previous sections.

Constructing a new block. Updating a ClaimChain implies creating a new block, and adding it to the chain. To create a new ClaimChain block, given the system security parameters

pp, the owner prepares:

(a) The claims to commit $\{..., (l_i, c_i), ...\}$, denoted as claims.

(b) The access control matrix acc, objects being labels that we denote here as l_i, and subjects being readers, represented by their public keys $\mathsf{pk}^A_{\mathsf{DH}}$.

(c) The cryptographic keys $(\mathsf{pk}_{\mathsf{DH}}, \mathsf{pk}_{\mathsf{SIG}}, \mathsf{pk}_{\mathsf{VRF}})$ and corresponding secret keys, denoted as keys.

(d) The signing key pair $\mathsf{sk}'_{\mathsf{SIG}}, \mathsf{pk}'_{\mathsf{SIG}}$, committed in the previous block (or initial signing key pair if the block is the genesis block).

(e) The references to previous blocks F (see 3.3.4).

Once these are set up, the owner proceeds as follows:

Step 1 Generate a nonce using the PRNG.

Step 2 Encode all claims using $\mathsf{EncodeClaim}$, producing a set of encoded claims $\{..., (L_i, C_i), ...\}$, and corresponding VRF values $\{..., k_i, ...\}$.

Step 3 For each row in the access control matrix, which represents a capability list for a reader, encode the capability entries using corresponding VRF values k_i using reader's $\mathsf{pk}^A_{\mathsf{DH}}$. The result is a set of capability entries $\{..., (L^{A,\ell}, P^{A,\ell}), ...\}$.

Step 4 Merge the sets of encoded claims and encoded capability entries into the set of map entries S.

Step 5 Build a non-equivocable Merkle tree T from the set of entries S, denoting the hash of its root node as MTR, using $\mathsf{BuildTree}$ procedure.

Step 6 Construct the block payload X (see 3.3.4) using the MTR as claim map hash, the public keys $(\mathsf{pk}_{\mathsf{DH}}, \mathsf{pk}_{\mathsf{SIG}}, \mathsf{pk}_{\mathsf{VRF}})$, timestamp, protocol version, and all the other additional information deemed relevant for the block meta-data.

Step 7 Sign the payload X using $\mathsf{sk}'_{\mathsf{SIG}}$, producing a signature σ.

Step 8 Obtain the block $B = \{X, \sigma\}$.

The owner can then put the block and the tree nodes in an Object Store (see Section 3.3.5). Depending on the ClaimChain operation mode (described in detail below, in Section 4), the owner may want to produce inclusion evidence for the relevant entries in the claim map for the respective readers using $\mathsf{ComputeEvidence}$.

Retrieval of the claim by label To be able to retrieve a claim from a block, the reader needs to parse the block, obtain the full claim map tree (or at least an evidence path containing the leaf with the claim). Given the system security parameters pp, the reader prepares:

(a) The claim label of interest $\mathsf{claim\_label}$

(b) The owner's public keys $(\mathsf{pk}_{\mathsf{DH}}, \mathsf{pk}_{\mathsf{VRF}}$ and the signing key from the owner's previous block $\mathsf{pk}'_{\mathsf{SIG}}$ (or current block if the block of interest is the genesis block).

(c) The hash of the owner's claim map tree root MTR.

(d) The block's nonce.

(e) A store containing the (sub-)tree T' of owner's claim map tree with all evidence paths to resolve the entries of interest.

The procedure to retrieve the claim ℓ with claim_label is following:

Step 1 Compute capability lookup key $L^{A,\ell}$ using the ComputeCapabilityLookupKey procedure.

Step 2 Query the tree with MTR for the lookup key $L^{A,\ell}$ using QueryTree procedure, getting the encoded capability P.

Step 3 Decode the capability P using the DecodeCapability procedure, obtaining the VRF value k, and deriving the claim lookup key L.

Step 4 Query the tree for the lookup key L. The result is an encoded claim C.

Step 5 Decode C and verify the VRF value using the DecodeClaim procedure.

Step 6 Recover claim_body.

4 ClaimChain as PKI

The ClaimChain structures provide system in which principals may dynamically create and update ClaimChains, embed meta-data about their own identities and keys into them, and certify the status of others' ClaimChains. These may be used to instantiate a decentralized, privacy-preserving, public key infrastructure, or even broader, federated identity system.

Overview & Semantics. A ClaimChain embeds some claims about the owner's information and claims about other ClaimChains' status. These claims can be given a precise meaning that enable clients to accept or reject bindings of names to keys, which in turn are necessary to support modern public key cryptography, from authentication to end-to-end encryption.

Consider a ClaimChain cc_A with meta-data A, and a claim binding name B to another ClaimChain cc_B (which we denote as $B \rightarrow cc_B$). Using the SecPAL language [18] formalism we can say this chain encodes the following two SecPAL statements:

- "cc_A says $A \rightarrow cc_A$" (self-claim)
- "cc_A says $B \rightarrow cc_B$" (cross-claim)

The ClaimChain system ensures that those basic claims about self of other users can be relied upon with high integrity. The structure of the chain further ensures that previous claims can be retracted, while new claims can be included in new blocks. Furthermore, the capability based access control system ensures that only authorized principals get access to claims at a given block.

4.1 Trust in ClaimChain claims

We consider that the owner of a ClaimChain implicitly trusts as true all statements that are included in their own chain. Thus, they can rely on the mappings between identities and ClaimChains to encrypt and sign to others. However, this is not necessarily true for claims that are present in other ClaimChain users' chains. For those to be trusted, a *social verification* process

needs to take place. We use the term social verification to emphasize that the rules under which users accept claims of other users *cannot* be universal or derived through deduction. Instead, each user (or her software) has to define personal rules that specify under what conditions claims in other chains can be trusted, be it self-claims or cross-claims.

Some examples of such *social verification* rules can be found in the literature:

- **Trust in certification authorities.** A user may chose to believe all statements that are included in one from a multitude of ClaimChains maintained by a set of certification authorities. This is equivalent to the model employed by web browsers can consider that a TLS certificate is authentic if it is used by one of a set of known certificate authorities.
- **Traditional web-of-trust.** A user may chose some of its 'friends' as being trusted, and will accept all claims that are included in their ClaimChains. This decentralizes the role of certification, in that different users may chose who has authority to make statements about others.
- **Threshold / social trust schemes.** A user may accept a binding between a name and a chain, or a self-claim, if a certain number of its 'friends' – other designated users – certify that claim. Thus, a single poor choice of 'friend' does not introduce the risk of incorrect bindings.

From a theoretical perspective, *social verification* is simply an algorithm that takes claims (either self-claims or cross-claims) and then resolves them into name-ClaimChain trusted bindings. This algorithm may be specified in a formal language, with well defined semantics and allowing for efficient resolution, such as SecPAL; or, it can simply be an ad-hoc algorithm specified in any computer programming language – with the risks or introducing inconsistencies or unexpected bindings in case of programming errors.

4.2 ClaimChain deployment options

We now describe two key deployment options, assuming that users have a social verification procedure in place.

On-line deployment. ClaimChain may be deployed in an on-line context, where we expect users trying to establish the veracity of claims using ClaimChains having the ability to query on-line services. In such a context, users can contact an on-line object store: to retrieve the latest state of another user's ClaimChain; to list all capabilities that they may access on such remote ClaimChain; and to retrieve all claims unlocked by those capabilities. As an optimization, a user may locally mirror some of the object store to avoid performing duplicate queries. In this model, when their own chain is updated through the generation of a new block, users need to upload the new state of the chain to the on-line object stores, to make it available for others to query the new blocks.

Besides on-line object stores, the on-line deployment setting can support providers that track the latest state of users' ClaimChains, and are ready to serve their latest block upon demand. Such providers need to do linear work, in the blocks uploaded to the store to maintain those indexes. However, clients only need to perform lookups to discover if ClaimChains of interest have been updated. Those providers need to be trusted to provide the latest updates to each chain. Multiple providers may be used, under the assumption that one of them is honest, to ensure no updates are delayed.

Asynchronous operation and embedding into Autocrypt [19]. Clients may have to operate off-line, without being able to query on-line services for ClaimChain-related information. A key example is the case of email communications, where emails may be composed or read off-line. Furthermore, it is considered good practice for Mail User Agents (MUAs) to not connect to services other than the email provider's for any checks. However, users asynchronously communicate with each other using email – which may be used to piggy-back some of the ClaimChain interactions.

In this deployment setting, each user of ClaimChain maintains a local off-line storage for objects, and their own ClaimChain. Users update their own local Object store when updating their chain; and use the 'natural' email communication with others to propagate the new state of their ClaimChains, and optionally others' ClaimChains, to their communication partners.

At a minimum we foresee users including in all their email communications the hash of their latest head, and potentially their latest block. This allows a user's communication partners to keep track of this user's ClaimChain, and updates to her signature and encryption keys. This presupposes frequent communications, and can be made more robust by including many previous blocks in case of infrequent communication to ensure recipients can authenticate the latest blocks from the sender.

With each message the sender may transfer the full capabilities and claims map, or just the meta-data. In the first case, recipients are able to access all cross-claims and rely on them to establish keys for others. In the second case, cross-claims are never available to others, and ClaimChain is reduced to only carry self-claims modeling the basic operation of Autocrypt [19]. Transferring all information with all emails is costly, despite the flexibility it offers.

ClaimChain can also support selective dissemination of claims to reduce the bandwidth overhead associated with each message. The latest block may be included with every message, and then only a selection of the capabilities and claims tree that is of relevance to the recipient. First, only the capabilities and claims that can be decoded and read by a recipient have to be included. Other information cannot be decoded, and thus would not be of direct use to the recipients. Secondly, if the sender foreseen only some claims being of use, a selection of those claims can be made available. This operation supports the recently developed Autocrypt Gossip model [20].

A key example relates to transferring only information useful to allow for introductions via email: a user Alice, writing an email to Bob and Charlie, may include in their message claims that would allow Bob and Charlie to check their own claims are in Alice's chain, and potentially to start relying on Alice's binding (if they trust Alice) to authenticate each others' identities. In that case Alice would include the latest blocks of Alice, Bob and Charlie, and also the capabilities and claims in her latest block that allow Bob and Charlie to establish that she has included their latest blocks are corresponding to their names. This option, reduces the fraction of the object store that needs to be send with each message – at the cost of allowing some claims to be resolved but not others.

In the case of email, the ClaimChains blocks and shards of the local object store can be encoded as email headers to be transferred transparently alongside messages. This provides the least obtrusive user experience on mail user agents that are not aware of ClaimChain. Alternatively, an attachment with ClaimChain, may be included in a multi-part SMTP envelope, with compliant clients being able to interpret it as ClaimChain information.

Finally, we note that an asynchronous query-response protocol may also be implemented: a compliant MUA can interpret special headers as requests for specific objects from the local store, and respond with an email containing the requested blocks. This has to be done with some care, as email users do not expect their MUA to be sending email without their knowledge, since this may leak presence information. To avoid such leakage, the queried blocks can be embedded into

future messages that the user sends to the requester.

5 Security and privacy definitions

In this section we formally define the security and privacy properties provided by ClaimChain. We prove that our constructions actually provide such properties reducing the security proofs to well-known properties of the underlying security primitives. Where needed we provide game-based definitions that help us illustrate the protection provided by ClaimChain.

5.1 Privacy

The ClaimChain system aims to provide a number of privacy properties, that have usually been missing from centralized or decentralized public key infrastructures. We consider that ClaimChain *privacy* is a composition of three different properties: *reader anonymity*, which ensures that the capabilities and claims contained in a ClaimChain containing do not leak the identity of those allowed to read those claims – guaranteeing the anonymity of the friends and contacts of the ClaimChain owner; *claim indistinguishability*, which ensures that a ClaimChain does not leak which claims it contains – protecting the privacy of the owner of the ClaimChain, and her social graph; and finally, *capability unlinkability*, which ensures that the mapping between readers of claims, and claims remains secret – preventing an adversary from inferring who has access to which claim. Each of those properties are defined separately, and proved cryptographically. We then compose them to formulate an overall *ClaimChain privacy* property.

Reader Anonymity. We first define a cryptographic game, called $\mathsf{ReadAnon}$ involving a polynomially bound adversary $\mathcal{A}(\cdot)$ and the procedures used to extend a ClaimChain cc_{n-1}.

Algorithm 8 The Reader Anonymity cryptographic game

function $\mathsf{ReadAnon}(\mathcal{A}, cc_{n-1}, \mathcal{C}, \mathcal{R})$
$r_0, r_1, R \leftarrow \mathcal{A}(\mathcal{R})$ $\quad (r_0 \neq r_1$ and r_0, r_1 not in $R)$
$\bar{s} \leftarrow \mathsf{sk}_r$ for all $r \in R$
$C, M \leftarrow \mathcal{A}(R, \mathcal{C})$
$b \leftarrow \{0, 1\}$
$cc_n \leftarrow \mathsf{CcExtend}(R \cup \{r_b\}, C, M)$
$b' \leftarrow \mathcal{A}(cc_n, \bar{s})$
return $b = b'$

In a nutshell this game accepts a universe of possible readers $\mathcal{R}$ and claims $\mathcal{C}$. The adversary then chooses two challenge readers r_0 and r_1, as well as a set of other readers R, a set of claims to extend a chain cc_{n-1}, and the mapping between all readers and claims M. The adversary is then given all secret keys $\bar{s}$ of all other readers R. A bit b is chosen at random and the ClaimChain cc_{n-1}, is extended with this data either using reader r_0 or reader r_1. The adversary is then provided the resulting ClaimChain, and has to infer the bit b, namely which challenge user was used to extend the chain.

Definition 1. *A* ClaimChain *provides* Reader Anonymity *if a PPT Adversary* $\mathcal{A}(\cdot)$ *may only win the* ReadAnon *game with negligible probability.*

$$\Pr[\mathsf{ReadAnon}(\mathcal{A}, cc_{n-1}, \mathcal{C}, \mathcal{R}) = 1] \leq \mathit{negl}(\kappa) \tag{4}$$

Proof. (Sketch) We note that the adversary does not know the secret DH key of the owner of the chain or either of the challenge readers. Thus, when the chain is created, the derived shared secret between the owner and either of the readers, would be indistinguishable to the adversary, according to the decisional Diffie-Hellman assumption. All of their computations after that are computational indistinguishable on the basis of this unknown shared DH key, and the identity of r_0 or r_1 is never used again. □

Claim Indistinguishability. Second, we define a cryptographic game, called CapInd, with the same inputs as the previous one.

Algorithm 9 The Capability Indistinguishability cryptographic game

function $\mathsf{CapInd}(\mathcal{A}, cc_{n-1}, \mathcal{R}, \mathcal{C})$
  $c_0, c_1, C \leftarrow \mathcal{A}(\mathcal{C})$ $(c_0 \neq c_1$ and c_0, c_1 not in $C)$
  $R, M \leftarrow \mathcal{A}(\mathcal{R}, \mathcal{C})$
  $\bar{k} \leftarrow k_e$ for all $c \in C$
  $b \leftarrow \{0, 1\}$
  $cc_n \leftarrow \mathsf{CcExtend}(C \cup c_b, R, M)$
  $b' \leftarrow \mathcal{A}(cc_n, \bar{k})$
  return $b = b'$

In this game, the adversary chooses two challenge claims c_0 and c_1, as well as a set of other claims C to extend the chain cc_{n-1}, a set of readers to which provide capabilities to read these claims, and the mapping between all readers R and claims M. The adversary is then given all secret keys $\bar{k}$ of all other claims C. A bit b is chosen at random and the ClaimChain is extended with either claim c_0 or claim c_1. The adversary is then provided the resulting ClaimChain, and has to infer the bit b, namely which challenge claim was used to extend the chain.

Definition 2. *A* ClaimChain *provides* Claim Indistinguishability *if a PPT Adversary* $\mathcal{A}(\cdot)$ *may only win the* CapInd *game with negligible probability.*

$$\Pr[\mathsf{CapInd}(\mathcal{A}, cc_{n-1}, \mathcal{C}, \mathcal{R}) = 1] \leq \mathit{negl}(\kappa) \tag{5}$$

Proof. (Sketch) We note that the adversary does not know the secret $\mathsf{sk}_{\mathsf{VRF}}$ key of the owner of the chain. Thus, she cannot guess the key k_e that would encrypt neither c_0 or c_1 with probability greater than it can forge the VRF signature on these claims labels and nonces, which is negligible. Thus we can substitute all calls to the VRF with a random VRF unknown to the adversary. Thus both keys for the claims are derived from random VRF elements and are from the same distribution, and under the CPA security of AES-GCM the resulting ciphertexts of the claims c_0 or c_1 are indistinguishable to the adversary. □

Capability Unlinkability. The third cryptographic game we define is called $\mathsf{CapUnlink}$, receives the same inputs as the previous games but takes the universe of all possible mappings claim-reader $\mathcal{M}$ instead of claims.

In this game, the adversary chooses two challenge mappings m_0 and m_1, that give access permission to claim c to either reader r_0 or r_1, a set of claims to extend a chain cc_{n-1}, and all other mappings between all readers and claims M. The adversary is then given all secret keys $\bar{s}$ of all other readers R. A bit b is chosen at random and the ClaimChain is extended with this data either providing access to c to reader r_0 or to reader r_1. The adversary is then provided the resulting ClaimChain, and has to infer the bit b, namely which challenge mapping was added to the chain.

Algorithm 10 The Capability Unlinkability cryptographic game

function $\mathsf{CapUnlink}(\mathcal{A}, cc_{n-1}, \mathcal{C}, \mathcal{R}, \mathcal{M})$
$m_0 = (c, r_0), m_1 = (c, r_1), R \leftarrow \mathcal{A}(\mathcal{C}, \mathcal{R})$
with ($c \in \mathcal{C}$, $r_0 \neq r_1$ and r_0, r_1 not R)
$\bar{s} \leftarrow \mathsf{sk}_r$ for all $r \in R$
$C, M \leftarrow \mathcal{A}(R, \mathcal{C})$
$b \leftarrow \{0, 1\}$
$cc_n \leftarrow \mathsf{CcExtend}(C \cup c, R \cup \{r_b\}, M \cup \{m_b\},)$
$b' \leftarrow \mathcal{A}(cc_n, \bar{s})$
return $b = b'$

Definition 3. *A* ClaimChain *provides* Capability unlinkability *if a PPT Adversary* $\mathcal{A}(\cdot)$ *may only win the* CapUnlink *game with negligible probability.*

$$\Pr[\mathsf{CapUnlink}(\mathcal{A}, cc_{n-1}, \mathcal{C}, \mathcal{R}, \mathcal{M}) = 1] \leq \mathit{negl}(\kappa) \tag{6}$$

Proof. (Sketch) We note that the adversary does not know the secret DH key of the owner of the chain or either of the challenge readers. Thus, when the chain is created, the derived shared secret between the owner and either of the readers, would be indistinguishable to the adversary, according to the decisional Diffie-Hellman assumption. All of their computations after that, including the publication of the mapping in the capability map are computational indistinguishable on the basis that the derived DH key is indistinguishable from a random element, and the CPA security of AES-GCM. □

ClaimChain Privacy. Finally, we compose the three properties above to define the privacy guarantees given by ClaimChain. To this end we define a game ClaimChainPriv that again involves a polynomially bound adversary $\mathcal{A}(\cdot)$ and the procedures used to extend a ClaimChain cc_{n-1}. The game accepts as inputs a universe of possible readers $\mathcal{R}$, claims $\mathcal{C}$.

Algorithm 11 The ClaimChain Privacy cryptographic game

function $\mathsf{ClaimChainPriv}(\mathcal{A}, cc_{n-1}, \mathcal{C}, \mathcal{R})$
$r_0, r_1, c_0, c_1, R, C \leftarrow \mathcal{A}(\mathcal{R})$
with ($r_0 \neq r_1$, and r_0, r_1 not R, $c_0 \neq c_1$, and c_0, c_1 not C)
$\bar{s} \leftarrow \mathsf{sk}_r$ for all $r \in R$
$M \leftarrow \mathcal{A}(R, C)$
$b \leftarrow \{0, 1\}$
$cc_n \leftarrow \mathsf{CcExtend}(C \cup \{c_b\}, R \cup \{r_b\}, M \cup \{(r_b, c_b)\},)$
$b' \leftarrow \mathcal{A}(cc_n, \bar{s})$
return $b = b'$

In this game, the adversary chooses two challenge pairs of readers and claims, r_0 and r_1, and c_0 and c_1, a set of claims to extend a chain cc_{n-1}, a set of readers to which provide capabilities to read these claims, and all other mappings between all readers and claims M. The adversary is then given all secret keys $\bar{s}$ of all other readers R. A bit b is chosen at random and the ClaimChain is extended with either claim c_0 accessible by reader r_0, or claim c_1 accessible by reader r_1. The adversary is then provided the resulting ClaimChain, and has to infer the bit b, namely which claim, reader, and access permission were added to the chain.

Definition 4. *A* ClaimChain *provides* ClaimChain Privacy *if a PPT Adversary* $\mathcal{A}(\cdot)$ *may only win the* ClaimChainPriv *game with negligible probability.*

$$\Pr[\mathsf{ClaimChainPriv}(\mathcal{A}, cc_{n-1}, \mathcal{C}, \mathcal{R}) = 1] \leq \mathit{negl}(\kappa) \tag{7}$$

Proof. (Sketch) The adversary cannot distinguish which reader is added due to the *Reader Anonymity* property, nor which claim is added due to *Claim Indistinguishability*, nor which mapping was added due to *Capability Unlinkability*. Therefore, the adversary cannot learn any of the information added to the chain and cannot win the game. □

5.2 Non-equivocation

A key property provided by ClaimChain is *non-equivocation*: given a claim in a ClaimChain, it is not possible for the owner to show different values of this claim to two different users. We define a cryptographic game, called NonEq involving a polynomially bound adversary $\mathcal{A}(\cdot)$ and the procedures used to create lookup keys, and to extend and read the tree containing the capabilities map.

Algorithm 12 The Non-equivocation cryptographic game

function $\mathsf{NonEq}(\mathcal{A}, \mathcal{C}, \mathcal{R})$
$cc \leftarrow \mathcal{A}(\mathcal{C}, \mathcal{R})$
$(r_0, l_0, c_0) \leftarrow \mathcal{A}(cc)$
$(r_1, l_1, c_1) \leftarrow \mathcal{A}(cc)$
$X_0 \leftarrow \text{Check}(cc, r_0, l_0, c_0)$
$X_1 \leftarrow \text{Check}(cc, r_1, l_1, c_1)$
return $X_0 \wedge X_1 \wedge (l_0 = l_1) \wedge (c_0 \neq c_1)$

In this game, the adversary builds an arbitrary structure that they pass as a ClaimChain. They then construct two sets of label-claims potentially to different readers (r_0, l_0, c_0) and (r_1, l_1, c_1). We return true (or 1) if those claims can both be checked as belonging in the ClaimChain, by the checking algorithm; they have the same label, but different contents for the claim.

Definition 5. *A* ClaimChain *provides* ClaimChain non-equivocation *if a PPT Adversary* $\mathcal{A}(\cdot)$ *may only win the* NonEq *game with negligible probability.*

$$\Pr[\mathsf{NonEq}(\mathcal{A}, \mathcal{C}, \mathcal{R}) = 1] \leq \mathit{negl}(\kappa) \tag{8}$$

Sketch: Non-equivocation relies on two properties of ClaimChain building blocks. First, Π_{VRF} guarantees that for a given claim_label known by the reader, and owner's VRF key pair $(\mathsf{sk}_{\mathsf{VRF}}, \mathsf{pk}_{\mathsf{VRF}})$, the adversary can only find one possible $k = \mathsf{VRF}_{\mathsf{sk}_{\mathsf{VRF}}}(\mathsf{nonce} \parallel \mathsf{claim\_label})$ with non-negligible probability, and therefore, a derived lookup key $L = H_1(k)$. Second, given the lookup key L and a hash of the non-equivocable Merkle tree root MTR, it is infeasible for the owner of a ClaimChain to present two different resolution paths that start in the same root but end in leaves with different content, without breaking the collision resistance property of the hash function used.

5.3 Integrity and authenticity

Block integrity. For a block B having imprint $H(B)$, it is unfeasible to find another block B' with claims, metadata, pointers to previous blocks, or any other data from the block B dropped, rearranged, or otherwise modified, such that $H(B) = H(B')$.

Proof. Trivial reduction to *second preimage resistance of* H. □

Extension to chains. For a chain $\mathbf{B} = \{B_1, B_2, ..., B_n\}$ with head $H(B_n)$, it is unfeasible to find another chain $\mathbf{B'} = \{B'_1, B'_2, ..., B'_m\}$ with claims, metadata, any other data from any block B_i of chain B, and blocks themselves dropped, rearranged, or otherwise modified, such that $H(B_n) = H(B'_m)$.

Proof. Consequence of hash chain construction and block integrity. □

Authenticity Given a block $B = (X,\ \sigma)$, where X is the payload of the block and σ its signature, and associated verification key pairs $(\mathsf{sk}_{\mathsf{SIG}}, \mathsf{pk}_{\mathsf{SIG}})$, an adversary cannot forge a new block.

Proof. Trivial reduction to selective unforgeability of the Π_S. □

6 Evaluation

We have evaluated ClaimChain scalability and efficiency. First, we measured the performance and storage (bandwidth) requirements of ClaimChain operations (see Section 6.2). Second, we also simulated the decentralized asynchronous deployment scenario from Section 4 using real data, and measured the speed of encryption keys and ClaimChain heads propagation using different modes of operation (see Section 6.3).

All the experiments in this section are reproducible, and are available as Jupyter notebooks [21] [22] in our code repository [23].

6.1 Implementation

For the purpose of conducting the experiments, we have implemented a ClaimChain prototype using Python [23]. For elliptic curve cryptography operations, this implementation uses George Danezis's petlib [24] library, which internally uses OpenSSL [25] C library. For the implementation of verifiable skip-list and sorted Merkle tree operations, we use George Danezis's hippiehug [26] library, which is written in pure Python.

In order to ease the implementation of ClaimChain, we slightly deviate from the specification of the block structure in Section 3.3.4. Due to the specifics of the hippiehug library, we "moved" the block index and the pointers to previous blocks outside of the block's payload, with corresponding changes to the signature. The block structure we use in the prototype is thus

$$B = \{\texttt{payload}, \texttt{index}, \texttt{pointers}, \sigma\},$$

where $\sigma = \mathsf{Sign}(\texttt{payload} \parallel \texttt{index} \parallel \texttt{pointers} \parallel \mathsf{None})$.

Cryptographic primitive instantiations For symmetric encryption, we use AES128 in GCM mode with IV fixed to an array of zeros (non-determinism comes from prepending a per-block nonce to every message to be encrypted). For public key cryptography, we use ECDSA, ECDH, and CONIKS VRF scheme [3], with a NIST/SECG curve over a 224 bit prime field. As a basic hash function we use SHA256, with H_1 through H_4 instantiated as SHA256 hash of message prepended with some prefix. All the lookup keys on the claim map are truncated to 8 bytes, which should ensure absence of collisions for up to 2^{32} entries in the map. The size of the per-block nonce is set to 16 bytes, using standard Linux `urandom` device as PRNG.

6.2 ClaimChain core operations performance

We have measured the computation time of the basic operations in ClaimChain, as well as bandwidth (or storage) size needed to keep and transmit the structures.

The experiments were run on a machine with an Intel Core i7-5600U CPU @ 2.60GHz using CPython 3.5.2. All the experiments assumed local in-memory storage. For the purpose of this evaluation, each chain block meta-data field (see Section 3.3.1) only includes cryptographic public keys.

Basic operations timing. We have measured the performance of the basic encoding and decoding ClaimChain operations from Section 3.4.1. To this extent, 1000 claims were encoded, each having content size of 512 bytes, and label size of 32 bytes. Both the contents and labels were randomly generated bytes. Additionally, 1000 readers were emulated, each having access to a single random claim, resulting in 1000 encoded capability entries. We then decoded each capability entry and the corresponding claims.

As discussed in Section 4, we assume that claims normally store heads of ClaimChains of other people. For some of the simulations, however, we also consider a scenario where cryptographic material is stored as claims. Since the content size of 512 bytes can fit an encoded 2048 bit RSA public key and some additional information, these timing measurements represent this worst-case scenario in which RSA cryptographic keys are stored as claims. The choice of of label size is because 32 bytes are sufficient to encode most common email addresses as ASCII strings. The number of 1000 claims and capabilities was chosen only to estimate the variance of computation time. Performance of all the basic operations does not depend on the number of claims in the block.

Table 1: ClaimChain basic operations timing

	avg (ms)	st. dev.
Single-label capab. lookup key computation	0.12	0.02
Single-label capab. decoding	0.14	0.02
Single-label capab. encoding	0.14	0.00
Claim encoding	1.51	0.16
VRF computation	1.46	0.16
Claim decoding	2.48	0.29
VRF verification	2.44	0.29

We report the average computation time and standard deviation of each basic operation per single claim or capability entry in Table 1. One can see that time to encode and decode claims is mostly the time of VRF computation and verification. Encoding and decoding of capabilities, and computing the capability lookup keys each take under 0.15 milliseconds, making the computation time basically negligible. The worst computation time is under 2.5 milliseconds for decoding a claim, which we consider fast enough for keeping ClaimChain clients efficient.

Constructing the claim map. The most computationally expensive operation that ClaimChain owners perform is constructing the claim map. Unlike basic operations, it depends on the number of entries. To measure the time to construct the claim map, we chose a range of numbers n_i from 100 to 5600, and for each n_i, we encoded n_i claims and n_i capability entries. As in the previous experiment, the way we produced these is generated and encoded n_i claims, and simulated n_i readers each having access to one random claim. For each number of claim-capability

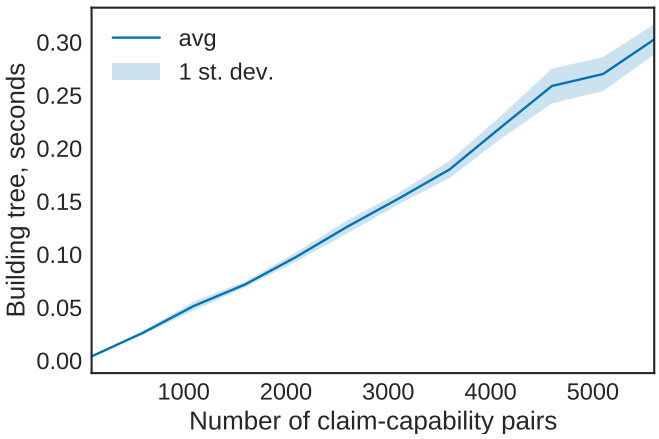


Figure 1: Claim map construction timing

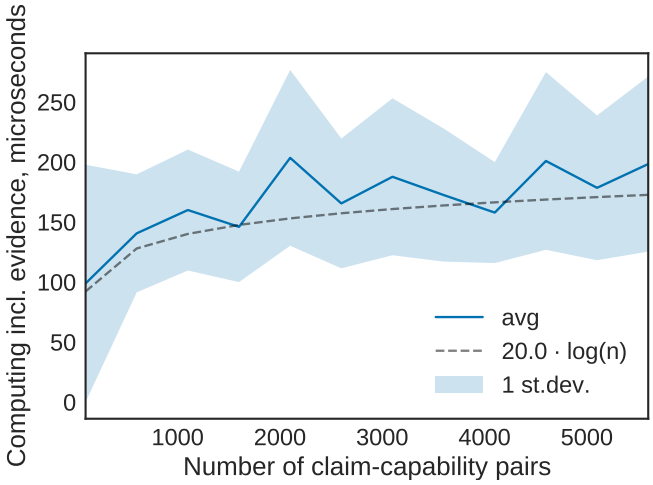


Figure 2: Computation time of inclusion evidence for a single entry

pairs in the range, a sorted Merkle tree containing the encoded entries was constructed 20 times. Only the time to build the tree itself was measured in this experiment, i.e. all the entries were encoded before the measurements started.

The reason why we report the results not in terms of the number of entries in the map, but rather in terms of claim-capability pairs, is the heterogeneity of claims and capability entries. The generated claims had a fixed content size of 512 bytes, while capability entries have different size, carrying a single VRF value . Recall that we are storing both claims and capabilities in the same tree for simplicity. By keeping the number of claims and capability entries equal in the experiment, we aimed to make the measurements balanced. Note that in practice, the number of capability entries would vary greatly depending on individual policies and social graphs. The number of 20 runs of building the tree was chosen to estimate the variance, while keeping the running time of our experiments reasonable.

The result can be seen in Figure 1. The operation takes under 0.3 seconds for 5000 claim-capability pairs. We expect average clients to have much fewer than 10000 map entries in their ClaimChains, as our further simulations in Section 6.3 demonstrate. Hence, the time of tree construction seems acceptable for scalable deployments of ClaimChain.

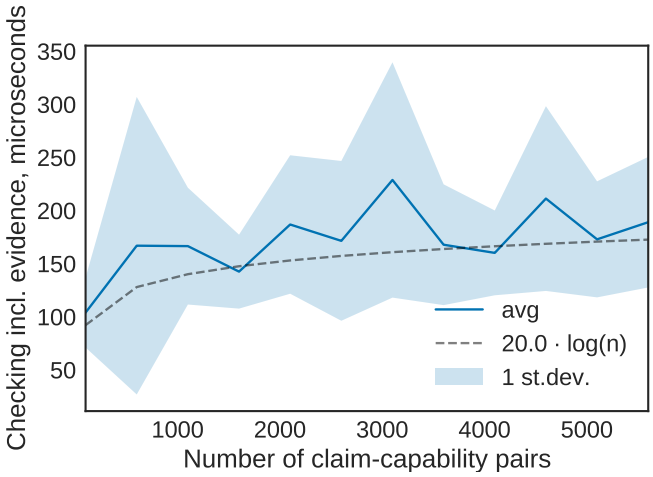


Figure 3: Verification time of inclusion evidence for a single entry

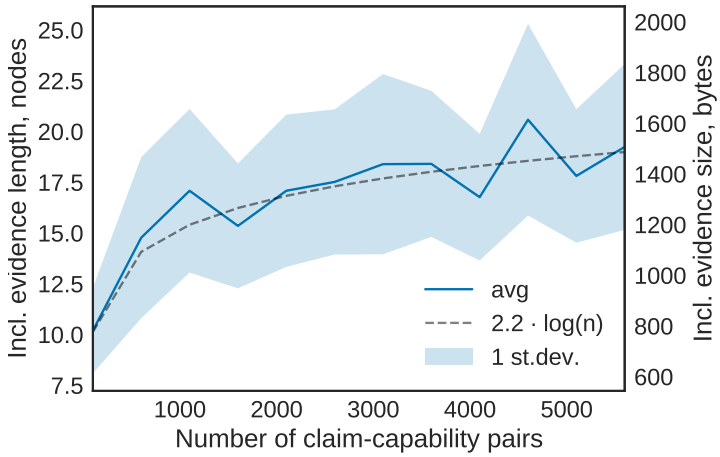


Figure 4: Size of inclusion evidence size for a single entry

Inclusion evidence measurements. In the asynchronous operation mode (see Section 4), we assume that along with the block, owners send relevant resolution paths in the claim map tree, that serve as evidence of inclusion of claims and capability entries in the tree. We measured the time to compute the inclusion evidence for a single map entry, and to verify a single piece of inclusion evidence, depending on the number of entries in the owner's claim map. We used the same setting as in the previous experiment, only after the tree was already constructed for each number of claim-capability pairs. For each n_i, 200 lookup keys from the claim map were chosen at random, and the evidence computation and verification times for these lookup keys were measured, as well as the evidence size. The number of 200 was chosen to estimate the variance in computation time and evidence size.

See Figure 2 and 3 for the timing results, and Figure 4 for the size measurements. The plots show that evidence computation time, verification time, and evidence size are all logarithmic in the number of entries in the map. Even though asymptotic complexity is logarithmic, computing many various evidence paths may prove too slow for the owner. In such a case, it could be worthwhile to trade off the computation time on the owner side for the bandwidth size by sending the whole tree (tree size measurements are reported further in this section).

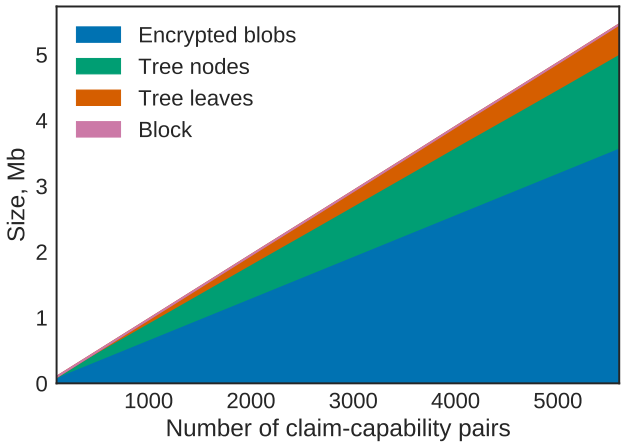


Figure 5: Storage size

ClaimChain storage size. To estimate the memory requirements for the owner to store a ClaimChain, we measured the size of the claim map tree, the values in the claim map ("encrypted blobs"), and a chain block size. The size of the claim map and size of all encrypted blobs depend on the number of entries in the map, and the size of encrypted blobs depends on the size of claims. We used the same setting as the previous two experiments, i.e. using claims with constant content size of 512 bytes and label size of 32 bytes, and varying the number of claim-capability pairs in the map within the range. Recall that after the claim is encoded, the lookup key in the claim map is 8 bytes per our parameter instantiation, not 32 bytes.

See the size breakdown depending on the number of claim-capability pairs in Figure 5. The block size is constant, and can only grow if security parameters change (size of cryptographic public keys, and hash length increase), or additional meta-data about the owner is added. We reported the block size in this graphic for completeness. One can see that the overhead of tree nodes is significant, but is under 2 megabytes even for 5000 claim-capability pairs, which suggests that sending the whole tree is a feasible trade-off for avoiding the computation of evidence paths in the asynchronous distributed setting.

6.3 The Enron email dataset

For the evaluation of information propagation using ClaimChain, we have used as input to our simulations data from the Enron Email Dataset [27, 28, 29]. This dataset is a dump of the email directories of 150 employees of Enron. In total, it contains around 500,000 emails. It has been widely used in previous research, for instance for email classification [30], or de-anonymization of social graphs in anonymous communications [31].

We construct the social graph of the 150 senders in the dataset by considering every receiver of a sender's email as a friend for this sender. In total, there are 20258 receivers, whose distribution as friends is shown in Figure 6. As we see in the figure, most of the senders, 30%, have a rather small social network (less than 50 friends), being the mean number of friends per user 150. However, there is a non-negligible tail of senders with large networks. A few users (8%) have more than 400 recipients, with one user having as much as 927 contacts.

For our evaluation, we have parsed a total of 517394 emails that we use to simulate traffic that generates changes and updates in their senders and receivers ClaimChains. Out of those, we have discarded 268681 emails because they were duplicate, 1330 that cannot be parsed, and

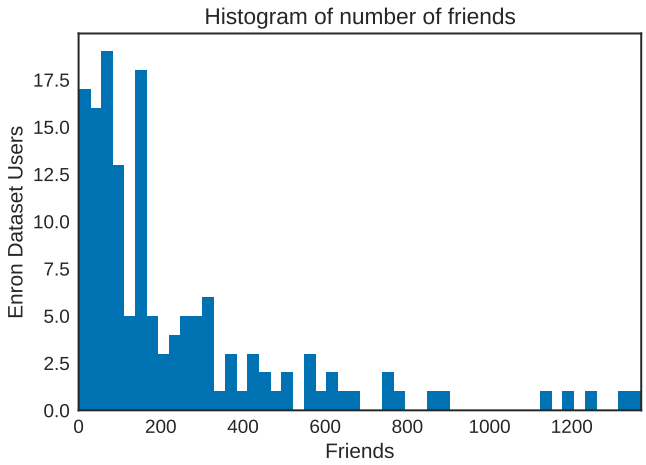


Figure 6: Social graph in the Enron Dataset: number of friends distribution

242 emails because they were lacking a valid recipient (IBM Lotus Notes format that cannot be mapped to the email address, or group of email addresses, it corresponds to).

Figure 7 illustrates the distribution of the number of recipients per email (which in turn gives an indication of the number of people that could receive ClaimChain updates in a decentralized scenario). As expected, we see that most emails (194873 out of the 245947) have less than five recipients. However, as in the case of the social graph the tail is long. We observe a non-negligible amount of emails with hundreds of recipients, even thousand for one particular case.

6.4 Simulation settings

In the following experiments, we study different configurations of ClaimChain. To this end we replay the full log of exchanged messages in a chronological order, and study the propagation of changes (heads or key updates) across users depending on the propagation model and privacy guarantees that users select when sharing their ClaimChains. Concretely, we study three models:

Basic Autocrypt – sender keys only [19] This model reflects the basic operation of Autocrypt in which Mail User Agents (MUAs) add specific headers in every outgoing email that include the latest encryption key of the sender, and her preference on receiving encrypted emails. Even in emails with multiple public recipients, a MUA only adds the encryption key of the sender. Therefore, a user can only learn of her friends' encryption key updates when she receives an email from them. Since users only propagate their own keys, that is the only information they store in their ClaimChains (i.e., they do not store claims about others' ClaimChains). Effectively this means that in this scenario keys and ClaimChain heads evolve at the same time.

ClaimChain without private claims This model studies ClaimChain under the assumption that users propagate their friends' latest status (heads and/or keys) as soon as they learn about them. Along with the status, users provide evidence for all public claims in their chain in every email they send. In turn, recipients parse the attached evidence and update their friends' entries accordingly. This scenario is analogous to the PGP Web of Trust, in which users always attach their public key along with signatures on the keys of their friends.

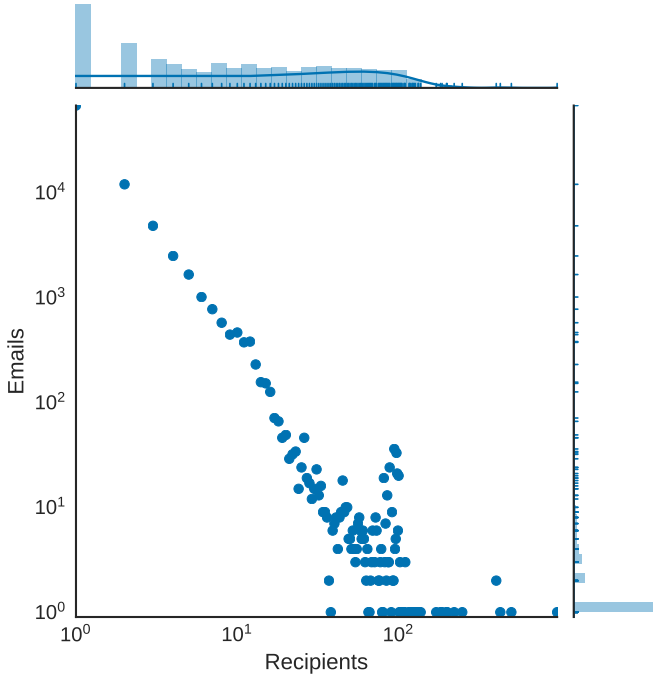


Figure 7: Recipients per email in the Enron Dataset

ClaimChain with capabilities and private claims Finally, we analyze the configuration of ClaimChain as described in this report. Senders implement access control to their ClaimChain, providing capabilities to recipients regarding the access to each other's updates. We build such access control lists in an incremental way. Every time there is an email with more than one recipients, all recipients earn the capability to see each other's update in the sender's ClaimChain. This capability is kept over time, i.e., recipients learn of updates of a common friend even if that friend is not included in the email thread. We note that this scenario could be adapted to simulate the Autocrypt Key Gossip feature [20]. This could be achieved by removing the capability map from the block, and sending in the header *only* the capabilities associated to claims related to the recipients of the email, as explained in Section 4.2.

In all experiments we assume that at the beginning of time ($t = 0$), all users update their encryption key and add a block on their ClaimChain. During simulations, ClaimChains get updated, i.e., a new block is appended, either when a user has 5 updates to be added (changes in a claim, or new claims), or when her own key is rotated.

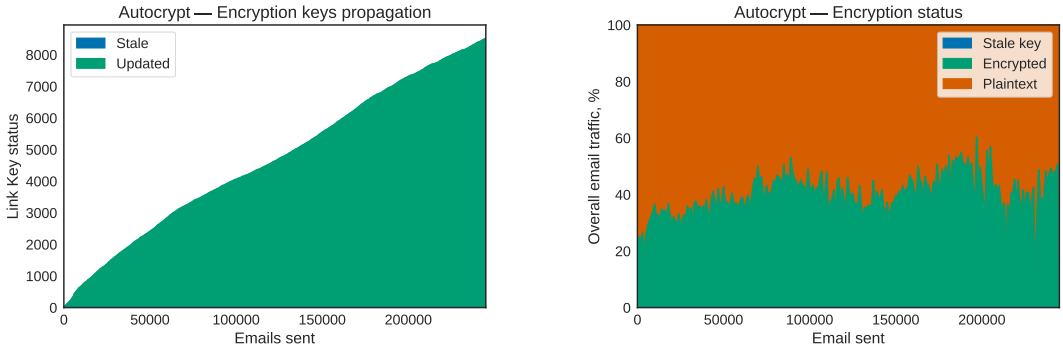
Furthermore, we consider that there is no high-availability storage to serve ClaimChain updates, but users discover the updates from embedings (such as special headers, or encrypted blobs) in the emails they exchange. We note that in a case with a high-availability storage key/head propagation is trivial, and therefore we do not study this scenario.

6.5 Static simulation scenario

Our first experiment studies the case where users state is static, i.e., they do not change their cryptographic key material over time.

6.5.1 Basic Autocrypt – sender keys only

The results for this scenario are shown in Figure 8. The top figures show the evolution of keys and heads that are propagated across links. These links represent unidirectional relationships from Enron employees to all their recipients (i.e., any other user to which they ever send an email) in the dataset. Since there are no key updates, we observe no stale keys in Figure 8a. As users only communicate their own keys, which are already known to other users, their ClaimChains do not evolve, and therefore key and head propagation are equivalent.



(a) Updated keys per link

(b) Percentage of encrypted emails

Figure 8: Static scenario: propagation in basic Autocrypt

In Figure 8b, we depict the percentage of emails that would be sent encrypted by employees of the Enron dataset. We compute this percentage over batches of 1000 emails over time, and we consider that an email is sent encrypted if at that point in time the sender knows the keys of all the receivers. First, we see that the number of encrypted emails (in green) does not monotonically increase. This is because users write to new people all the time, and since they have not received the keys in the past, they cannot encrypt their emails. Yet, we see that over time the number of encrypted emails increases, up to 50% at the end of the simulation.

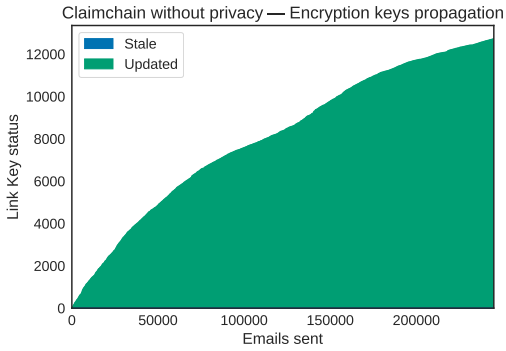
6.5.2 ClaimChain without private claims

When users also include information about others in their ClaimChains we observe two effects. First, the number of keys propagated is much larger by about 4000 links (see Figure 9a). Second, as opposed to the previous scenario, since users learn about new user's status and add them to their ClaimChains, their ClaimChains evolve. Thus, even though propagation is much more effective than before in absolute numbers, many of these heads become stale after some time. This affects the possibility to validate other users' ClaimChains, but we note that any learned key is correct (in this scenario keys do not rotate, thus they do not become stale).

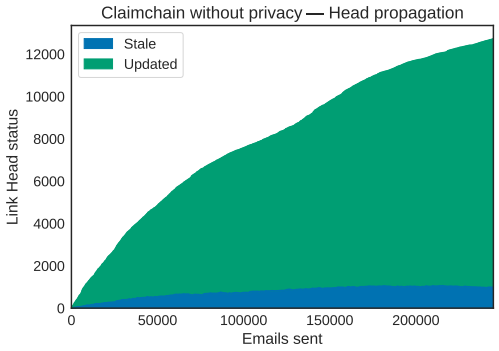
Although, as in the previous case, sharing more keys does not necessarily means better security, the improvement in key propagation shows positively in Figure 9c, where we see how in general the number of encrypted emails increase. Yet, due to the fact that users write to new recipients continuously, plaintext emails do not disappear.

6.5.3 ClaimChain with private claims

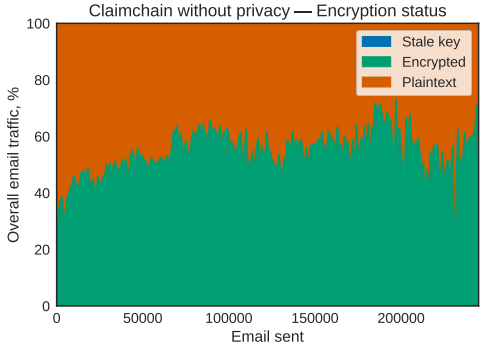
The advantage in terms of propagation when using ClaimChain is reduced when users implement access control to preserve the privacy of their social graph. Due to the slow introduction policy that we establish (i.e., capabilities are only added when users are recipients of the same



(a) Updated keys per link



(b) Updated heads per link



(c) Percentage of encrypted emails

Figure 9: Static scenario: propagation in ClaimChain without private claims

email), this case shows a very similar behaviour to that of Autocrypt in terms of key propagation, as shown by Figure 10a and by Figure 10c. In this case, 8807 keys are propagated, only 291 more than the 8516 updated keys in Autocrypt. Note, however, that due to introductions ClaimChains heads do change. Hence some of the users' knowledge about other users' heads becomes stale over time.

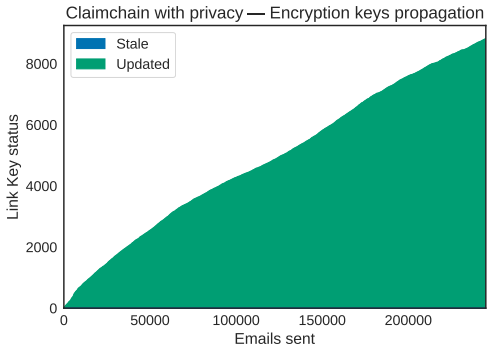
6.6 Dynamic simulation scenario

In this experiments we consider that users change their keys as time goes by. For the sake of illustration we chose as trigger event for key rotation the sending of 50 messages.

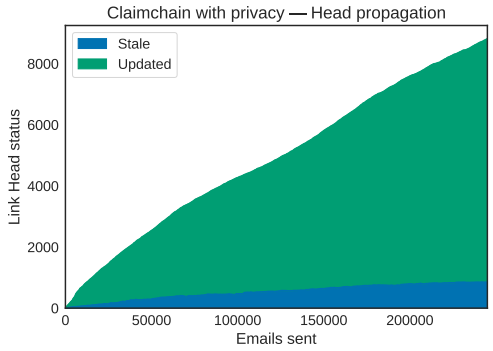
6.6.1 Basic Autocrypt – sender keys only

We observe in Figure 11 a similar behaviour to the static simulation, the difference being that, due to key rotation, some of the keys and heads that users have learned become stale over time. The appearance of stale keys on users' ClaimChains is reflected in the security of the communications network. As opposed to Section 6.5.1, in which whenever a sender knew the key of her receiver this resulted on a securely encrypted email, now we observe that a fraction of emails get encrypted with stale keys.

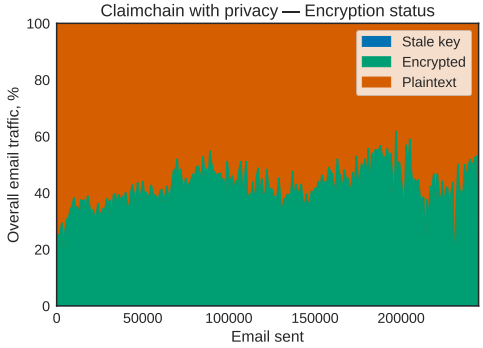
The use of stale keys does not compromise the security of the system with respect to third party eavesdropper. However, it may be problematic if keys have become stale because of key theft, i.e., the adversary that has the key can read those encrypted emails; or if the recipient also has lost access to the key and thus can neither read the email even though she is in principle



(a) Updated keys per link



(b) Updated heads per link



(c) Percentage of encrypted emails

Figure 10: Static scenario: propagation in ClaimChain with private claims

authorized to do so. We note that, if there is no theft or loss, users can keep old keys after rotation to make sure that they can read any email they receive.

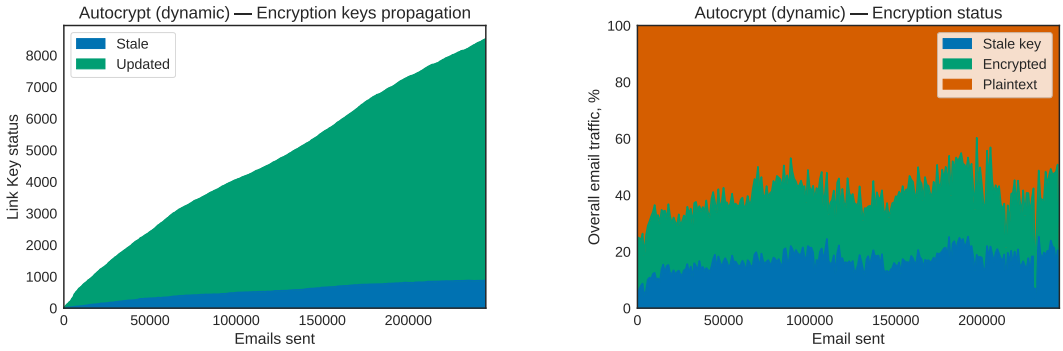
6.6.2 ClaimChain without private claims

We see in Figures 12a and 12b that, as in the static case, enabling recipients to read claims about other users in the ClaimChain improves key and head propagation with respect to the basic Autocrypt case. On the negative side, better propagation results in a slight increase of the number of stale keys, and respectively, heads. This is due to the fact that users learning about friends of friends does not guarantee that they will hear again about them.

However, we note that the fact that users end up with many stale keys does not reflect the security of communications. As we see in Figure 12c, though the amount of stale keys is larger than in basic Autocrypt, the amount of encrypted emails still increases. This is because frequent communication partners keep their keys up to date.

6.6.3 ClaimChain with private claims

Finally, when access control is implemented we again see little difference with respect to Autocrypt in terms of keys and head propagation over links, due to the introduction policy. This similar propagation behavior is also reflected in the number of encrypted emails in Figure 13c.



(a) Updated keys per link

(b) Percentage of encrypted emails

Figure 11: Dynamic scenario: propagation in basic Autocrypt

7 Conclusions

Public key infrastructure plays an integral role in enabling secure communications. However, most existing PKI implementations are centralized, and as a result, PKI providers need to be trusted by users for acting honestly. Moreover, these providers are in a privileged position to conduct surveillance on users.

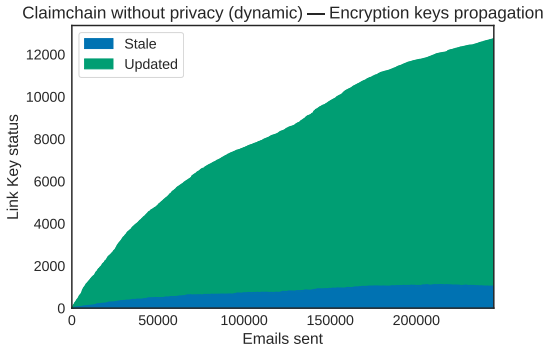
We have presented ClaimChain, a PKI design in which every user or device maintains repositories of claims regarding their own and their contacts' public keys. The high integrity of these repositories is maintained by virtue of storing claims on authenticated data structures: hash chains and Merkle trees. We have introduced the concept of cross-referencing of hash chains, and described how cross-referencing can be used to build a decentralized PKI to establish trust in identity-key bindings, and detect compromises. As a result, our PKI can *operate in fully decentralized setting* with no trusted parties, yet it is *flexible* enough that it can be deployed in semi-trusted centralized, federated, and hybrid modes. We have provided a way to *preserve privacy* of claims while at the same time *preventing users from equivocating*. We have evaluated the design in terms of performance of basic operations, bandwidth requirements, and key material propagation effectiveness in the fully decentralized setting.

7.1 Future work

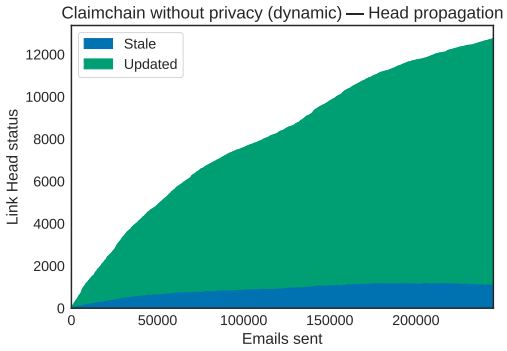
We identify the following directions for the future work:

Design of social validation policies. We have outlined the basic framework for the functioning of social validation policies in ClaimChain. However, for real-world deployment, the policies need to be precisely and formally defined and evaluated in terms of their security and effectiveness.

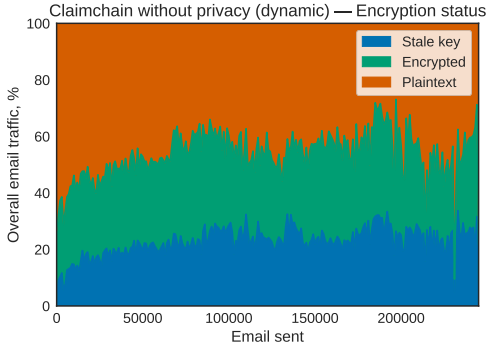
Quantifying the privacy leakage of social validation policies. The mere fact of ClaimChain user Alice being able to use Bob's latest encryption key may leak information about Alice's social graph. Imagine that Bob only has one friend Carol, with whom he shares his ClaimChain state. If Bob has not communicated with Alice before, yet at some point receives an encrypted message from her, he may conclude the Alice must know Carol. Thus, even though we ensure ClaimChains themselves do not leak any information about the social graph of users,



(a) Updated keys per link



(b) Updated heads per link



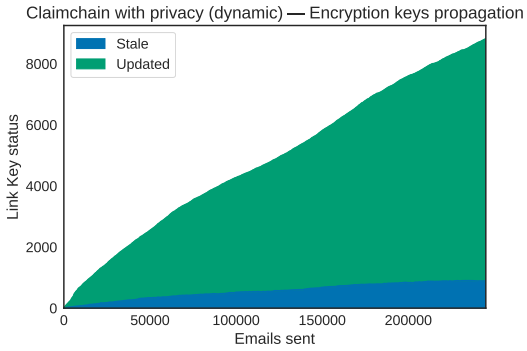
(c) Percentage of encrypted emails

Figure 12: Dynamic scenario: propagation in ClaimChain without private claims

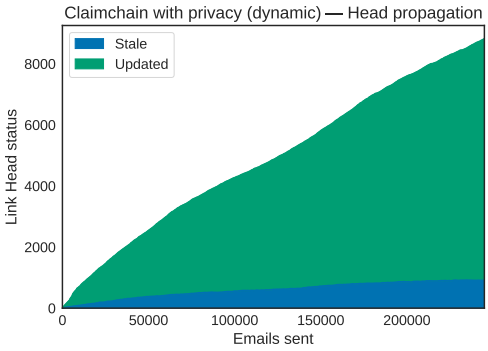
the social validation outcomes do. The social validation policies, therefore, have to be formally evaluated in terms of such privacy leakage.

Protocols for key revocations and chain compromises. The design of ClaimChain ensures that chain compromises are evident as forks of hash chains. It is, however, unclear what is the best way to deal with detected compromises, and what are the possible trade-offs in terms of security, privacy, and efficiency. ClaimChain supports generic claims, which could include statements useful in this context, e.g. chain recovery statements using secret information assumed to be uncompromisable, and signals to other users about detected forks or revoked keys. Precise protocols need to be devised to deal with these potentially dangerous situations, likely as parts of social validation policies.

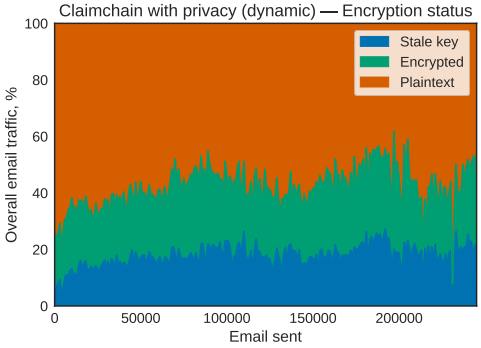
Evaluating scalability and effectiveness, and usability. We have evaluated ClaimChain on a real-life data set, but evaluation *in real world* may reveal unforeseen challenges for the design or the implementation. Besides that, more work is needed to understand the user experience and usability aspects of the ClaimChain. PGP has long been criticized [32] for its usability issues, hindering its truly widespread usage; more recently, Marcela Melara has identified [33] that usability-related issues were the biggest challenge for deploying CONIKS in the real world. Without further investigation, it is not clear if ClaimChain suffers from similar problems, and what exactly needs to be improved or changed.



(a) Updated keys per link



(b) Updated heads per link



(c) Percentage of encrypted emails

Figure 13: Dynamic scenario: propagation in ClaimChain with private claims

Transferability of access control tokens. As is the case with traditional capability-based access control systems, access control tokens in ClaimChain (VRF values k_i) are transferable. Transferability may be undesirable, since malicious users may pass the access tokens to parties that were originally not authorized to read the claims. We were unable to find a way to make these tokens non-transferable without sacrificing either non-equivocation, or privacy. This lead us to believe it may be impossible to satisfy all three properties simultaneously. Proving that this statement is true, or otherwise, devising a scheme in which access tokens are non-transferable, non-equivocable, and private at the same time, merits further research.

References

[1] "SKS Keyservers." `https://sks-keyservers.net`. Last accessed: July 18, 2017.

[2] C. Adams, S. Farrell, T. Kause, and T. Mononen, "Internet x. 509 public key infrastructure certificate management protocol (cmp)." RFC 4210, 2005.

[3] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: bringing key transparency to end users," in *24th USENIX Security Symposium, USENIX Security 15* (J. Jung and T. Holz, eds.), pp. 383–398, USENIX Association, 2015.

[4] P. R. Zimmermann, *The official PGP user's guide*. MIT press, 1995.

[5] "PGP Global Directory." `https://keyserver.pgp.com`. Last accessed: July 18, 2017.

[6] "Mailvelope." https://keys.mailvelope.com. Last accessed: July 18, 2017.

[7] "Nyms Identity Directory." http://nyms.io. Last accessed: July 18, 2017.

[8] "Nicknym." https://leap.se/en/docs/design/nicknym. Last accessed: July 18, 2017.

[9] B. Laurie, L. A., and E. Kasper, "Certificate transparency." RFC 6962, June 2013.

[10] E. Kokoris-Kogias, L. Gasser, I. Khoffi, P. Jovanovic, N. Gailly, and B. Ford, "Managing identities using blockchains and CoSi," in *9th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2016)*, 2016.

[11] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping authorities "honest or bust" with decentralized witness cosigning," in *IEEE Symposium on Security and Privacy, SP 2016*, pp. 526–545, IEEE Computer Society, 2016.

[12] M. Ali, J. C. Nelson, R. Shea, and M. J. Freedman, "Blockstack: A global naming and storage system secured by blockchains.," in *USENIX Annual Technical Conference*, pp. 181–194, 2016.

[13] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[14] "Keybase.io." https://keybase.io. Last accessed: July 18, 2017.

[15] J. I. Munro, T. Papadakis, and R. Sedgewick, "Deterministic skip lists," in *Proceedings of the Third Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms.* (G. N. Frederickson, ed.), pp. 367–375, ACM/SIAM, 1992.

[16] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Commun. ACM*, vol. 19, no. 8, pp. 461–471, 1976.

[17] R. R. Toledo, G. Danezis, and I. Goldberg, "Lower-cost $\in$-private information retrieval," *PoPETs*, vol. 2016, no. 4, pp. 184–201, 2016.

[18] M. Y. Becker, C. Fournet, and A. D. Gordon, "Secpal: Design and semantics of a decentralized authorization language," *Journal of Computer Security*, vol. 18, no. 4, pp. 619–665, 2010.

[19] "Autocrypt: E-Mail Encryption for Everyone." https://autocrypt.org/. Last accessed: July 18, 2017.

[20] "Autocrypt Level 1: Enabling encryption, avoiding annoyances." https://autocrypt.org/en/latest/level1.html#key-gossip. Last accessed: July 18, 2017.

[21] F. Pérez and B. E. Granger, "IPython: a system for interactive scientific computing," *Computing in Science and Engineering*, vol. 9, pp. 21–29, May 2007.

[22] "Jupyter project." https://jupyter.org. Last accessed: July 18, 2017.

[23] "claimchain-core – a core and experimental implementation of the ClaimChain distributed PKI." https://github.com/gdanezis/claimchain-core/. Last accessed: July 18, 2017.

[24] "petlib – a python library that implements a number of Privacy Enhancing Technolgies." "https://github.com/gdanezis/petlib". Last accessed: July 18, 2017.

[25] "OpenSSL – cryptography and ssl/tls toolkit." https://openssl.org.

[26] "hippiehug – a verifiable skip-list and sorted Merkle tree implementation." https://github.com/gdanezis/rousseau-chain/tree/master/hippiehug-package. Last accessed: July 18, 2017.

[27] "Enron email database." http://www.cs.cmu.edu/~enron/. Last accessed: July 18, 2017.

[28] B. Klimt and Y. Yang, "Introducing the Enron Corpus," in *CEAS 2004 - First Conference on Email and Anti-Spam*, 2004.

[29] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[30] J. Shetty and J. Adibi, "Discovering important nodes through graph entropy the case of enron email database," in *3rd international workshop on Link discovery, LinkKDD* (J. Adibi, M. Grobelnik, D. Mladenic, and P. Pantel, eds.), pp. 74–81, ACM, 2005.

[31] G. Danezis and C. Troncoso, "You cannot hide for long: de-anonymization of real-world dynamic behaviour," in *Workshop on Privacy in the Electronic Society, WPES 2013* (A. Sadeghi and S. Foresti, eds.), pp. 49–60, ACM, 2013.

[32] A. Whitten and J. D. Tygar, "Why johnny can't encrypt: A usability evaluation of pgp 5.0.," in *USENIX Security Symposium*, vol. 348, 1999.

[33] M. Melara, "Why making johnny's key management transparent is so challenging." https://freedom-to-tinker.com/2016/03/31/why-making-johnnys-key-management-transparent-is-so-challenging/. Last accessed: July 18, 2017.