NEXt generation Techno-social Legal Encryption Access and Privacy nextleap.eu

Grant No. 688722. Project started 2016-01-01. Duration 36 months.

DELIVERABLE D4.4

Privacy Preserving Messaging Scalability Simulations

Carmela Troncoso, Wouter Lueks, Bogdan Kulynych

Beneficiaries:	**EPFL (formerly IMDEA)** (lead)
Workpackage:	**WP4: Validation via Formal Modeling and Simulation**
Description:	This deliverable summarizes the work on scalability of messaging protocols performed within WP4. As the scalability and content security of the NEXTLEAP secure messaging protocols was established in other deliverables, we focus on technical developments oriented to improving the anonymity and privacy of communication and key-related operations. We present LoopTor and Lightnion, two anonymous communications modules that can be easily deployed on top of the Tor network. We include an improved version of Claimchain that prevents a new attack identified in WP5. Finally, we propose a protocol that enables users to securely perform cryptographic operations in untrusted devices.
Version:	Draft
Nature:	Report (R)
Dissemination level:	Public (P)
Pages:	88
Date:	2019-1-17

Project co-funded by the European Commission within the Horizon 2020 Programme.

Contents

1 Introduction **4**
1.1 Anonymity on the network layer . . . 4
1.2 Security and Privacy in key distribution . . . 5
1.3 Security and privacy in key use . . . 6

2 LoopTor **6**
2.1 Introduction . . . 6
2.2 System . . . 7
2.2.1 Goals and non-goals . . . 7
2.2.2 An overview of LoopTor . . . 8
2.3 Building LoopTor . . . 8
2.3.1 One link to rule them all . . . 9
2.3.2 Back to the 90s . . . 9
2.3.3 Do clog the queue! . . . 9
2.3.4 Implementing LoopTor . . . 10
2.4 Evaluation . . . 12
2.4.1 Preliminary measurements . . . 12
2.5 Conclusions . . . 12

3 Lightnion **13**
3.1 Introduction . . . 13
3.2 Background . . . 15
3.3 System . . . 16
3.3.1 Parties and trust assumptions . . . 17
3.4 Architecture . . . 17
3.5 Evaluation . . . 18
3.5.1 Implementation . . . 18
3.5.2 Performance . . . 18
3.5.3 Next steps . . . 20
3.6 Conclusions and future work . . . 20

4 Claimchain **20**
4.1 Introduction . . . 21
4.2 Problem statement and goals . . . 22
4.3 ClaimChain design . . . 23
4.3.1 Cryptographic preliminaries and notation . . . 23
4.3.2 Overview . . . 24
4.3.3 Low-level operations . . . 25
4.3.4 High-level operations . . . 28
4.3.5 Security and privacy properties . . . 31
4.4 Using ClaimChains to secure in-band key distribution . . . 32
4.4.1 Detecting inter-block equivocation . . . 34
4.5 Evaluating the performance of ClaimChain . . . 35
4.5.1 Experimental setup . . . 35
4.5.2 ClaimChain operations performance . . . 36
4.6 Concluding remarks . . . 37

5 Tandem **38**
5.1 Introduction . . . 38
5.2 Related Work . . . 40
5.3 Problem Statement . . . 41

5.3.1 TANDEM Properties and Threat model . . . 41
5.3.2 TANDEM at a Glance . . . 43
5.4 Cryptographic preliminaries . . . 44
5.4.1 Cryptographic Building Blocks . . . 45
5.4.2 Threshold-Cryptographic Protocols . . . 45
5.5 TANDEM . . . 46
5.5.1 One-time-use Key-share Tokens . . . 47
5.5.2 Alternative constructions . . . 52
5.6 Security and Privacy of TANDEM . . . 53
5.6.1 Security of TANDEM . . . 53
5.6.2 Privacy of TANDEM . . . 55
5.7 Securing protocols with TANDEM . . . 56
5.7.1 The use-case of ABCs . . . 57
5.7.2 Rate-limiting in ABCs . . . 59
5.8 Performance Evaluation . . . 60
5.9 Conclusion . . . 62

6 Conclusion **63**

References **63**

A Claimchain security and privacy **68**
A.1 Unique-resolution key-value Merkle tree . . . 68
A.1.1 Algorithms . . . 68
A.1.2 Unique resolution . . . 69
A.2 Security of the ClaimChain data structure . . . 71
A.2.1 Privacy . . . 71
A.2.2 Non-equivocation . . . 75

B Tandem security and privacy **78**
B.1 Proofs of Lemmas . . . 78
B.2 Constructing Correctness Proof of $\overline{x_S}$ . . . 78
B.3 Security Proof . . . 82
B.4 Privacy Proof . . . 84

1 Introduction

The NEXTLEAP partners have proposed two decentralized private messaging protocols designed within WP2 and WP5: The MLS messaging protocol described in deliverable D2.3 and D4.4, and the Autocrypt protocol described across all deliverables in WP5 together with D2.4. The security and privacy properties of these protocols have been thoroughly analyzed in those deliverables. The properties of MLS have been proven using strict formal methods. The new Autocrypt proposal, being more complex and difficult to fully model, has been carefully examined by the NEXTLEAP partners as well as subjected to scrutiny by the security community.

The goal of task 4.2 is to evaluate the scalability of the NEXTLEAP private messaging protocols, focusing on their capabilities to provide decentralization, privacy, security, and anonymity. The three first properties, however, are intrinsic to the nature of the final design of the protocols. For instance, Autocrypt has been built on top of the email infrastructure, which is decentralized by nature. Autocrypt is already integrated by a number of email clients, and by DeltaChat (an email-based instant messaging application), to provide end-to-end encryption for their users. The current pilots demonstrate that the protocols scale without a problem. In the case of MLS, while keeping the centralized nature of Signal, it has been designed to be *more* scalable than Signal's double ratchet messaging protocol. In terms of security and privacy, as mentioned above, both protocols have gone through a thorough evaluation. Thus, we have focused our efforts on producing modules that can improve the anonymity properties of these systems, as well as on improving the security and privacy of tasks complementary to the exchange of messages such as key storage and exchange or key usage while preserving the decentralized nature of NEXTLEAP protocols.

1.1 Anonymity on the network layer

While MLS and Autocrypt offer strong confidentiality for messages, they cannot by themselves provide anonymity. If an adversary observes the communications between users and the server (the email or messaging provider) the identities of communication partners are revealed. In order to provide anonymity these protocols must rely on anonymous communications networks to hide the relationships among users, as well as their communication partners.

In this work-package we have developed two anonymous communication modules aimed at augmenting the privacy of the messaging protocols. Instead of relying on novel anonymous communication systems that may or may not get massive deployment [Pio+17], we choose to use the widely deployed anonymous communication network Tor. Tor [DMS04a] is the most widely used anonymous communication network. At any point in time, millions of users use Tor to anonymously communicate on the Internet.[1] Tor is an overlay anonymity network that provides anonymity to users by routing their traffic through different Tor nodes. Currently, the Tor network consists of more than 6000 nodes operated by volunteers. The NEXTLEAP modules do not require any changes in the thousands of volunteer nodes that make up the Tor network, and thus can immediately be integrated by applications to provide stronger anonymity for many protocols, including messaging.

Our modules tackle two properties that the current Tor deployment fails to address.

[1] `https://metrics.torproject.org/userstats-relay-country.html`

The first issue we address is that Tor is vulnerable to traffic correlation attacks. The design of Tor is optimized to reduce the bandwidth and latency overhead imposed by the anonymous communication layer, so that Tor is suitable for a large variety of tasks such as browsing the Internet, messaging, e-mail, and peer to peer data sharing. The decision of keeping Tor low-latency at a low bandwidth overhead opens the door to traffic correlation attacks that enable network attackers to determine who communicates with whom by correlating traffic entering and existing the Tor network.

Tor assumes that such an adversary is unlikely to exist. This is a reasonable assumption when the communicating parties are located in geographically diverse locations, because in that case the attacker needs to observe a large fraction of the Internet to correlate traffic. In messaging, however, the sender and recipient are often in close geographic proximity (i.e., in the same country). Therefore, it is plausible that within this geographic location a network adversary exists that can observe traffic entering and exiting the Tor network (e.g., a national law enforcement authority, or a national Internet Service Provider).

In Section 2 we introduce the *LoopTor module*. This module enables Tor clients to resist traffic correlation attack at a low cost for low-throughput application such as messaging. LoopTor builds on top of the existing Tor network without requiring any modification and thus can be directly deployed. We show through simulation that LoopTor mitigates the capability of an adversary to perform traffic correlation attacks.

A second issue with the current Tor deployment is usability. To securely enjoy the advantages Tor provides, users need to access the Tor network via the Tor Browser. This hinders adoption of this solution by service providers, since requiring an extra browser to use a service is bound to reduce their user base. Efforts such as Protonmail illustrate the importance of offering solutions that are easy to use, for example, by developing secure messaging clients that operate from a user's web browser. However, connecting a web-based client to the Tor network to enhance its privacy properties is currently non-trivial due to the complexity of the Tor network.

In Section 3 we introduce Lightnion, a small Javascript client that can be used to make anonymous requests from web sites, without requiring users to install any custom software. Thereby, Lightnion enables the development of easy to use web-based messaging clients that provide strong anonymity guarantees when, for example, connecting to a key server, or other third parties.

1.2 Security and Privacy in key distribution

To send encrypted messages users need to know the keys of their communicating parties. In D2.2 we showed that simply gossiping these keys, as Autocrypt version 1 proposes, leaks a lot of information about a user's social graph. To protect this information we proposed a secure privacy-preserving key-sharing method approach based on Claimchains[2], the NEXTLEAP federated identity building block. Using Claimchain, users can selectively give access to their contacts' keys. The protocol design proposed in D2.2 ensures that users cannot equivocate within a Claimchain block. However, during the integration of Claimchains into Autocrypt version 2, we realized that the original protocol allows malicious users to abuse the access control mechanism to equivocate accross blocks. In Section 4 we present an improved

[2] https://claimchain.github.io

version of the Claimchain protocols to make this attack detectable. Moreover, we complemented the informal security arguments in the previous version with formal security proofs.

1.3 Security and privacy in key use

An effect of decentralization is that critical parts of the protocols, and in particular the cryptographic keys, move to the users' devices. This move allows users to act without needing to interact with or obtain permission from central parties, thereby improving privacy. However, to maintain the security and privacy properties of decentralized protocols, the keys on the users' devices must be kept secure. If they are not and an adversary gets access to the keys they might impersonate users, spend their (digital) cash, or read their encrypted emails. The security of users' devices is, sadly, often limited, putting users' keys at risk.

A common approach to avoid that a user's device becomes a single point of failure for security and privacy is the use of threshold cryptography. Instead of relying on a single key, users split their key into different parts in such a way all of these parts are required to use the key. The user then stores these parts at different locations. Threshold cryptography thereby mitigates the risk of a device compromise: an attacker might obtain one part of the key, but will not obtain the other parts needed to actually use the key. While threshold cryptography provides strong secrecy regarding the keys themselves, it does not hide the very action of using the key. In particular, the user must contact the key share holders every time she wants to perform an operation, revealing the existence of this operation to these entities. In Section 5 we propose a scheme that permits users to hide their actions from key holders, while at the same time preventing key holders from abusing the keys.

2 LoopTor

The secure messaging protocols designed within NEXTLEAP, MLS and Autocrypt, provide strong security and privacy properties for message content. However, they cannot by themselves provide users with anonymity against network adversaries. In this section we introduce LoopTor, an anonymous communication system that helps to protect messaging systems against strong network adversaries that can observe both sides of the (anonymous) communication channel.

2.1 Introduction

Tor is a general-purpose anonymous communication network. It is low-latency to support a wide variety of applications, including web browsing, at a low bandwidth overhead. On the downside, this decision makes Tor vulnerable to traffic-correlation attacks. Passive observers that can observe traffic entering and exiting the Tor network can determine who communicates with who.

There are two main solutions for this issue [Das+18]. Either add significant delays in the communications to destroy timing patterns (e.g., traditional mix networks such as Mixmaster [Mul+] or Mixminion [DDM03]); or add a large amount of cover traffic to hide real communications patterns (e.g, modern mix networks such as Loopix [Pio+17]). These solutions, however, either require the deployment

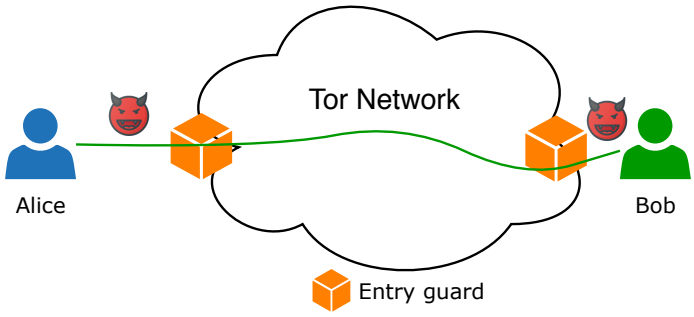


Figure 1: LoopTor's system model. A user Alice communicates with a another user Bob via the Tor network. Both users use LoopTor. The adversary can observe the network traffic between the client and the Tor network and between the server and the Tor network.

of new infrastructure, or impose a high overhead on the users and the network. As a result, they are inconvenient and difficult to deploy.

In this section we introduce LoopTor, a medium-latency communication scheme built on top of Tor that protects users against powerful adversaries that can observe traffic entering and exiting the Tor network. LoopTor is designed with the two following goals in mind to ease deployability:

- *Minimal changes to the infrastructure*: the design must not require any modification in the onion routers, only on the client side that is under the control of the users, so that LoopTor can fully benefit from the existing Tor infrastructure.
- *Minimal overhead on the network*: the design must not impose a high cost in terms of bandwidth or processing for the onion routers, so that its use does not affect other Tor users.

To achieve this goals we need to increase latency and decrease throughput. While this choice may seem controversial, we observe that many applications can actually tolerate medium latency and low throughput. For example, messaging, e-mail, transactions of digital currencies, and electronic voting, can tolerate some delay and do not have high bandwidth requirements.

2.2 System

In LoopTor a client, Alice, communicates with a server, Bob, via the Tor network. The adversary can observe the network traffic between Alice and the Tor network and between Bob and the Tor network. See Figure 1.

2.2.1 Goals and non-goals

LoopTor aims to provide sender and receiver unobservability [Pio+17] against adversaries that can observe the traffic between the users, e.g., Alice or Bob, and the Tor network. More precisely, we aim to achieve the following two properties:

Sender unobservability A network adversary observing the traffic between an online sender S (i.e., Alice) and the Tor network cannot decide whether S is currently communicating with any receiver (i.e, Bob) or not.

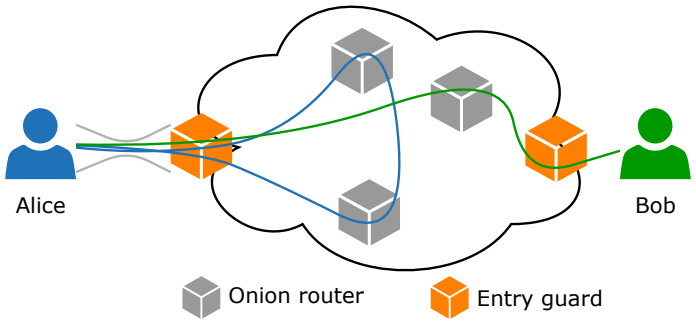


Figure 2: Overview of LoopTor. Alice (blue) uses LoopTor to protect her connection with Bob (green) against traffic analysis. LoopTor throttles the link with the entry guard and fills this link using dummy loop traffic (blue).

Receiver unobservability A network adversary observing the traffic between an online receiver R (i.e., Bob) and the Tor network cannot decide whether R is currently communicating with any sender (i.e., Alice) or not.

We require that the sender respectively receiver is online. If they are not, then obviously, that user cannot send respectively receive.

LoopTor routes dummy traffic and real traffic through the Tor entry node. An adversary that can distinguish these can break the unobservability properties of LoopTor. Therefore, we assume that the attacker has limited capabilities:

- The attacker is passive, in particular it cannot corrupt Tor entry nodes; and
- The attacker cannot observe traffic between the Tor entry node and the rest of the Tor network.

2.2.2 An overview of LoopTor

LoopTor builds on two key insights. The first insight is that if the link between the user and the Tor network were very low bandwidth, think 90s modem speed, flooding the link with dummy traffic should have low impact on the Tor network. The second insight is that users can generate dummy traffic by creating a connection to themselves, and sending dummy packages on this link [Pio+17]. This loop traffic ensures that, without help from the network or other users, the link can be easily saturated with traffic in both directions. See Figure 2 for an overview of LoopTor.

2.3 Building LoopTor

To ease deployment, LoopTor aims to use the existing Tor network. Therefore, its design cannot require any changes to the thousands of volunteer nodes that make up the Tor network. Such a requirement on the core Tor infrastructure is unlikely to be deployed. Changing software that connects to the Tor infrastructure, i.e., the Tor clients running on users' machines, however, is feasible. These user-side modifications do not require changes in the global infrastructure and can be performed locally by interested entities.

In this section we explain how we use functionalities provided by existing Tor nodes to construct the link between the client and Tor network, how we slow it down, and

how we generate loop traffic.

2.3.1 One link to rule them all

LoopTor users send and receive both dummy loop traffic and real traffic. Our network adversary can monitor the network between the user and the Tor entry guard. As a result, the adversary should not be able to distinguish these two types of traffic. Normally, however, Tor creates a new TCP connection to Alice for every incoming connection (including the loop connection). Hence, by just observing the number of incoming connections a passive adversary can learn if and when Alice is communicating: precisely what LoopTor aims to hide.

Thus, to be safe, LoopTor must route all virtual connections, both incoming and outgoing, via one TLS connection between the user and the entry guard. To implement this without changing any Tor protocol, LoopTor resorts to an existing Tor primitive that already achieves what we want: a hidden service. Every user sets up a hidden service to accept incoming traffic. As a result, all incoming traffic from other users is automatically routed via the single TLS connection between the user and the entry guard. LoopTor makes sure to use the same TLS connection to create the loop connection back to its own hidden service, see Figure 3. As a result, all LoopTor traffic is routed via one TLS connection.

2.3.2 Back to the 90s

To artificially create a slow connection between the user and the Tor network, we employ TCP's existing congestion control algorithms to limit the speed at which the entry guard sends traffic to the user. In particular, we use the existing network tool tc[3], to set the incoming bandwidth limit for each LoopTor user.

To signal to the entry guard that it should send less traffic, tc simply drops packets. This causes some overhead, as the entry guard does eventually resend those packets at a slower rate, thus increasing bandwidth usage, and creating a delay in the time it takes for packets to reach their destination.

Telling the entry guard directly to send traffic at a slow rate is more effective, as it would bypass TCP's congestion control algorithms. However, currently Tor nodes do not support this feature, and since LoopTor aims to make no changes to existing nodes, we cannot use this feature.

2.3.3 Do clog the queue!

Once the rate limit is imposed, sending dummy packets on the loop from the user back to herself will clog the queue, ensuring that the entry guard always has queued messages waiting to be sent to the user. To ensure that sufficient messages are available in the queue, LoopTor starts by sending a sizable number of dummy packets that prime the entry guard's buffer. Whenever LoopTor receives a dummy packet, it sends a new one to keep the buffer filled.

Since all communication with the entry guard is encrypted, attackers cannot distinguish dummy traffic from real traffic. Moreover, since regardless of traffic from other circuits there are always messages waiting to be sent to the user, the traffic

[3] http://man7.org/linux/man-pages/man8/tc.8.html

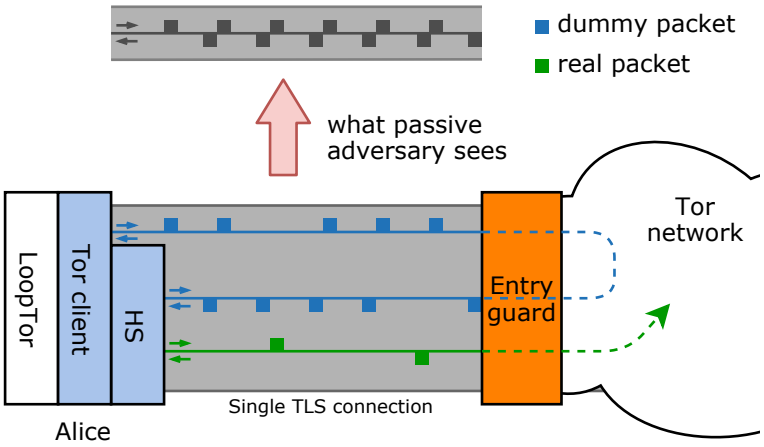


Figure 3: Detailed view of LoopTor showing the loop connection and one client connection. Both of these use the same TLS link to the entry guard. At the top: the adversary's view of this connection.

patterns observed by the passive adversary do not correlate to traffic sent by other users. See Figure 3.

While this construction prevents traffic-correlation attacks, the difference in incoming and outgoing traffic can still reveal whether a user is communicating. Loop traffic is symmetric by construction: every outgoing dummy packet eventually returns to the sending user. However, real traffic needs not to operate that way. To ensure that real and dummy traffic are indistinguishable, LoopTor ensures that for every incoming real message, it sends an equal-sized reply.

To summarize, an attacker will always see an independent number of indistinguishable messages regardless of the number of real or dummy messages being sent.

2.3.4 Implementing LoopTor

We implemented a LoopTor client in Python[4]. The client builds on top of the existing Tor client to connect to the Tor network. As we described in the previous section, the LoopTor client sets up a hidden service, publishes it, and then connects to its own hidden service to start generating loop traffic.

We have kept the changes to the Tor client to a minimum in our LoopTor prototype. However, one change was inevitable. Recall from Section 2.3.1 that all traffic between the user and the entry guard must share the same TLS connection between the user and the guard. The default Tor client tries to use an additional Tor guard when it detects that the current one is unavailable or too slow. In LoopTor, the guard is slow *by design*. Therefore, if we use an unmodified Tor client, multiple TLS connections would be created simultaneously, contradicting the requirement that all traffic shares the same TLS connection. As a consequence, we needed to modify the Tor client used by LoopTor to *never* switch guards.

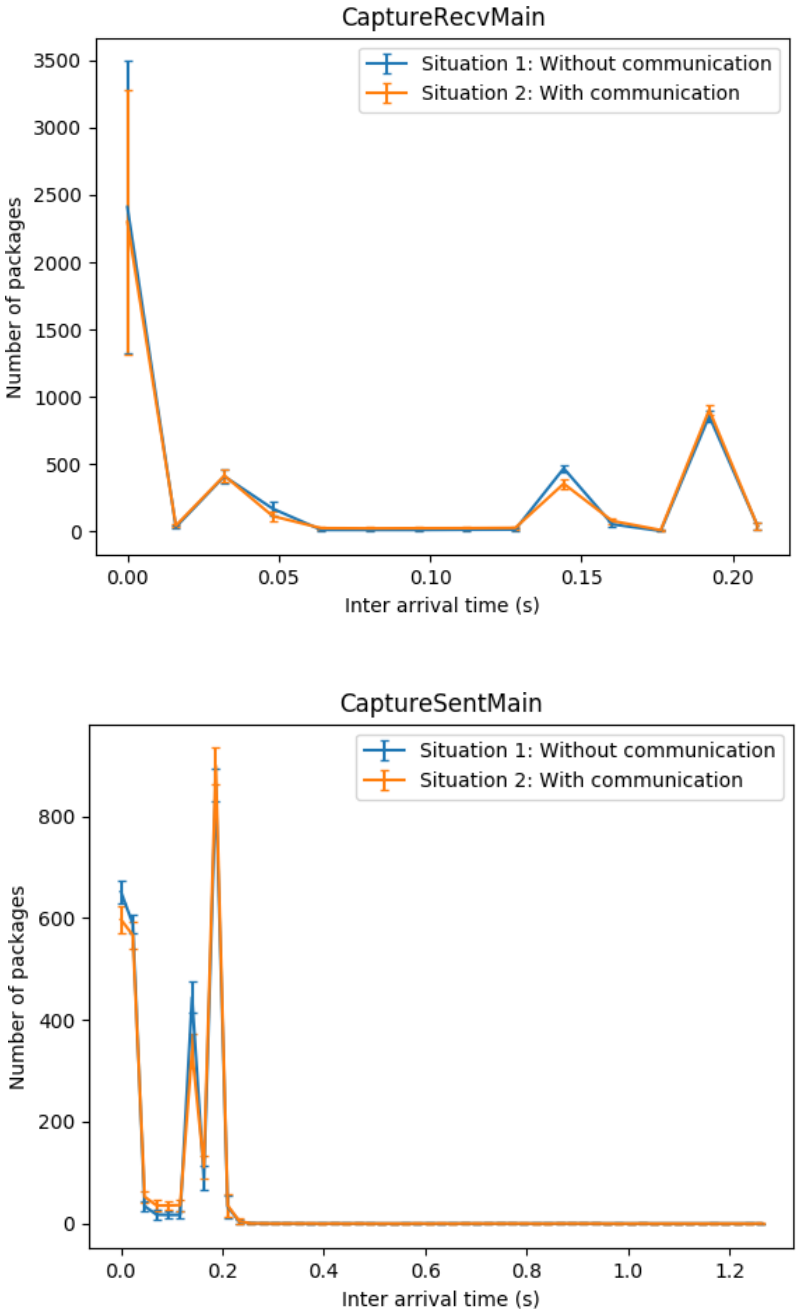


Figure 4: Inter-arrival time distributions for packages received by the observed user Bob (top figure) and sent by the user Bob (bottom figure). The graphs show the distributions for the two situations we compare: without communication (i.e., padding only), and with communication to other users. The error-bars show 95% confidence intervals. For a perfectly functioning LoopTor, the distributions should be statistically close. For now, some significant differences between the two distributions remain.

2.4 Evaluation

The goal of LoopTor is to ensure sender and receiver unobservability against passive network attackers. We conduct the following experiment to evaluate that these properties hold. A user, say Bob, uses LoopTor. The adversary monitors the connection between Bob and the Tor network. The task of the adversary is to distinguish two situations:

1. Bob is *not* communicating with any other users. As a result, all traffic observed by the adversary is dummy traffic.
2. Bob is communicating with other users using LoopTor. To simulate this, in the experiment new users regularly connect to Bob and send a message. The rate at which messages are sent is well below that of the cover traffic rate.

In order to test the worst-case scenario we deliberately let many users repeatedly communicate with Bob. As a result, the situations are easier to distinguish than in the setting where Bob only receives one message.

2.4.1 Preliminary measurements

We ran 50 experiments of 6 minutes length for each of the situations, and captured the network traces as they would be observed by the adversary. We then applied statistical measurements to compare the two situations. Figure 4 compares the inter-arrival time distributions for incoming and outgoing packages of Bob in the two situations.

The graphs confirm that the situations are close. Unfortunately, we still observe differences that may allow an adversary to distinguish the case in which Bob is communicating from the case in which he is not. We also observe that when Bob communicates with actual clients, the number of sent and received messages goes down.

We conjecture that these differences are caused by how incoming traffic is handled in both situations. When a new user connects to Bob it sets up a hidden-service connection to Bob. When setting up a hidden service, packages are answered directly by Bob's Tor client, without involving any LoopTor code written in Python. As such, some the responses are sent faster than when packages need to be first handled by LoopTor code. We are working on mitigating these issues in the next version of LoopTor.

2.5 Conclusions

We have introduced LoopTor, a module that can be used on top of the *existing* Tor network to provide a medium-latency low-bandwidth anonymous communication channel that resists passive network adversaries observing both ends of the communication channel. Such adversaries are likely to exist when users are messaging other users that are geographically close. Therefore, we believe that LoopTor is a very important complement to the messaging protocols developed within NEXTLEAP.

[4] We will make the code of LoopTor, as well as the code to reproduce our evaluations publicly available under an open source license at `https://github.com/spring-epfl/looptor`.

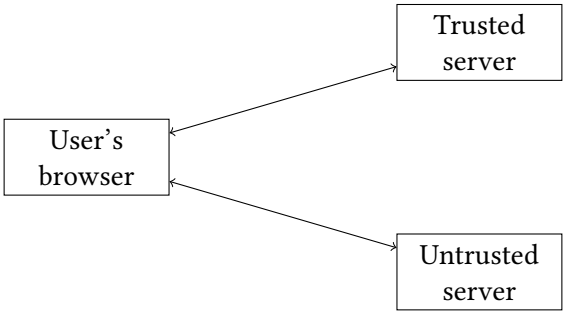


Figure 5: Setting of trusted and untrusted parties

3 Lightnion

In this section we tackle another problem of existing anonymous communication systems: They are often difficult to integrate into existing software, and in particular into software provided by online service providers. Therefore, in this section we propose a new system, Lightnion, a small JavaScript client that can be used to make anonymous requests from existing websites.

Both within and outside NEXTLEAP we have seen tools push for security by default and as easy as possible to use for users. For example, Autocrypt aims to automate security as much as possible, and only ask security related questions when users expect them. Lightnion allows us to build online platforms and websites that follow a similar paradigm for anonymity. Providers of platforms and websites can integrate Lightnion to seamlessly provide network anonymity for users without requiring users to do take any special actions.

3.1 Introduction

Users use Tor for a large variety of purposes, e.g., to browse websites, to share files, to send email, to use instant messaging services, or to visit hidden services (websites that are not accessible via the normal Internet). These different applications have widely different characteristics. E-mail requires low bandwidth and can tolerate a lot of latency, and messaging is low bandwidth and tolerates some latency. Both require only a few TCP connections. On the other hand, file sharing and web browsing, especially when visiting streaming websites, require a large number of concurrent TCP connections, some of which require high bandwidth and low latency.

The Tor client has been optimized to work well in all of these scenarios. It supports a large number of possibly long-term and high-bandwidth connections, and in addition to the normal protocols to communicate anonymously with existing servers it also supports protocols to enable hidden services. All these features complicate the Tor client. Most users experience Tor via the Tor Browser, a special browser that focuses on user anonymity and that offers tight integration with the Tor client. The Tor Browser, however, is not effortless for ordinary users. It requires them to install special software, and remember to use it instead of their normal browser whenever they require anonymity.

Using the Tor Browser ensures that all web activities from the user enjoy anonymity at the network layer. However such all-encompassing anonymity is not always necessary. Consider the following scenarios. In these, strong anonymity is only required for particular actions or against particular third parties, see also Figure 5.

- **Secure webmail.** Consider users using a webmail client for encrypted end-to-end communication. Users trust the provider of their webmail client to provide a secure service and software. However, the webmail client may need to interact with external, untrusted services. For example, to encrypt an email, the client needs to find the keys of the recipients. To do so, the webmail client in the user's browser makes requests to a key server. The user trusts the webmail client to know about her actions, but not the key server (who learns the communication partners of the user by virtue of seeing the key requests).
- **Accessing medical resources.** Consider doctors accessing the results of a patient's medical tests using a trusted online platform or email. To provide extra information about the results, the trusted platform refers to untrusted external websites. When a doctor visits these external sites, these sites might learn which doctors, and/or patients, suffer from certain diseases. The doctor trusts the online platform, but not the external websites.
- **Privacy-preserving collaborative editing.** Consider a privacy-preserving collaborative editing service. The platform provides collaborative editing software. This software uses cryptography to ensure that the platform does not know the content of the documents. Users trust, or verify the software provided by the editing platform. However, the editing platform might also learn who edits which sensitive files. Users may trust the software provided by the editing platform, but they do not necessarily trust the provider to know which files they edit.
- **Anonymous questionnaire website.** Consider an online questionnaire dealing with sensitive personal questions. To ensure the privacy of participants, the questionnaire platform does *not* want to be able to link the answers to individual users. This linking can occur in two ways: (1) via data submitted back to the server, and (2) via information implicitly sent at the network layer as users submit answers. The platform can easily ensure that no explicit identifiers are submitted together with the answers. However, the network layer identifiers remain. Users trust the questionnaire website, but they do not want to provide any identifying information when submitting their sensitive answers. Nor does the questionnaire website want to every have the user's identifiers for a set of answers in any form.

A centralized strawman approach. In all these scenarios users interact with both trusted platforms and untrusted servers. One way to ensure that users can anonymously access the untrusted server is to use a *centralized* and trusted proxy server. This proxy server would mediate the communication between the user and the untrusted server. As a result, the untrusted server only sees the network identifier of the proxy, and not those of the user.

However, trusted proxy servers suffer from several problems. First, operating a proxy server requires effort and expertise, and puts the operator of the proxy server at risk [New96]. Second, the proxy server and the untrusted server must not collude, as together they can break the user's anonymity. The webmail provider, and the trusted medical platform could themselves provide a trusted proxy to their clients, if they are willing to put in the effort to provide the infrastructure. However, the editing platform and the questionnaires site cannot. Doing so violates the non-collusion assumption. Instead, they must find a non-colluding third party that is willing to act as a proxy.

Decentralizing anonymity. To alleviate these problems we propose Lightnion, a web-friendly library for decentralized anonymous communication services. Using Lightnion, any trusted service provider can let their users communicate anonymously with untrusted third parties. Lightnion uses the existing Tor network to provide anonymity for users. It consists of a small Javascript library that trusted providers can embed in their websites. Websites can then take make anonymous connections to untrusted servers using the Lightnion library. To do so, the library connects to the Tor network via an untrusted proxy, and then sets up an anonymous connection via the Tor network to the untrusted server.

In this section, we make the following contributions:

- We analyze the trust-assumptions required for an anonymous communication library in the browser.
- We apply this analysis to derive a minimal Lightnion library that uses the Tor network and protocols.
- We constructed a prototype implementation of Lightnion and analyze its performance.

We note that while our efforts have been so far put in constructing a minimal, yet secure, Tor client for the web setting, the design principles and most of the code can be reused in other non-browser settings (e.g., to enhance mobile applications with anonymity).

3.2 Background

Tor is an overlay anonymity network consisting of many Tor nodes, or Onion Routers (OR). Users can anonymously communicate with a server by routing their traffic through a sequence of onion routers called a *path*. By default, the Tor client uses a path of three routers, called the guard, middle and exit nodes respectively. Each router on the path only knows the identity of the two adjacent nodes. As a result, none of the Tor nodes know both the user and the destination server. Traffic between the Tor client, running on the user's machine, and nodes in the Tor network is onion encrypted. Each subsequent node along the path removes one layer of encryption, until finally the exit node removes the last layer and forwards the traffic to the destination server.

The Tor network consists of many Tor nodes, each with different capabilities and available bandwidth. A group of nine Tor directory authorities periodically (every hour) publish a *consensus* of all the nodes in the network. Tor nodes themselves publish *micro descriptors* describing their keys and other properties. Tor clients use the consensus to pick a path and the descriptors to authenticate Tor nodes.

To create an anonymous connection to a destination server, the Tor client proceeds as follows.

1. The client retrieves the latest consensus and verifies the signatures by the trusted directory authorities. It also retrieves micro descriptors for many Tor nodes.
2. The client uses the consensus information to compute a path consisting of three nodes: the guard, middle and exit nodes.

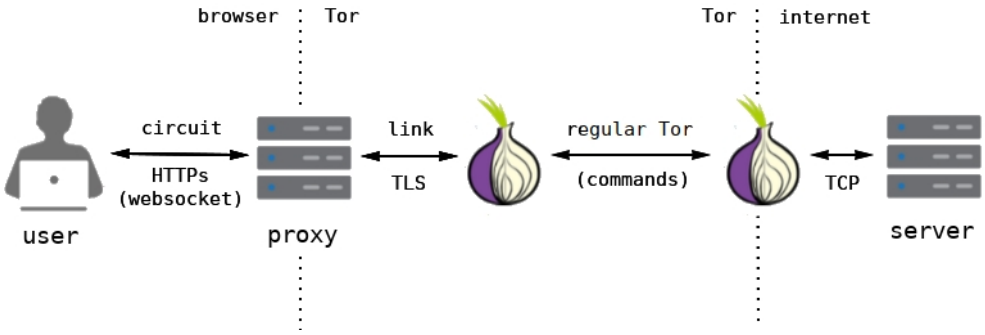


Figure 6: High-level architecture of Lightnion.

3. The client makes a TLS connection, called a link, to the guard node. The client authenticates the guard based on its micro descriptors and derives a shared key. The guard and the client use this shared symmetric key to encrypt subsequent traffic between then. Finally, the client starts building a circuit.
4. The client then requests the guard node to extend the circuit to the middle node. The client then authenticates the middle node (using the information from the micro descriptor) by running a handshake protocol. The result is a new symmetric key that the client and the middle node use to encrypt their traffic.
5. The client requests the middle node to extend the circuit to the exit node. The client and exit node run the hand shake protocol again to derive a final key.
6. Finally, the client requests the exit node to open a TCP connection to the target server.

Challenges in implementing a Tor client in the browser. Implementing a simple Tor client directly in the browser is challenging. To connect to the Tor network, a Tor client opens a TLS connection to the guard over TCP/IP. However, browser scripts can only create HTTP and Websocket connections. Pure TCP/IP connections are not allowed.[5]

Instead, in this work we use a simple, untrusted Lightion proxy that translates between a protocol that is available to the Javascript Lightnion client, such as HTTP or Websockets, and the TCP protocol that Tor nodes understand.

3.3 System

Figure 6 shows a high-level overview of the Lightnion architecture. Lightnion consists of two parts. The first part is a Javascript library that runs inside the user's browser. Websites can use this library to make anonymous web requests via the Tor network. To do so, the Lightnion Javascript client connects to a Lightnion proxy, the second Lightnion component, via a Websocket connection. The proxy relays Tor cells it receives over the websocket connection to a corresponding TLS connection that it maintains with guards in the Tor network.

[5]There exist some hacks such that circumvent these restrictions on some browsers, for example using WebRTC, Flash plugins or Java applets. However, none of these are portable or supported in a wide range of browsers.

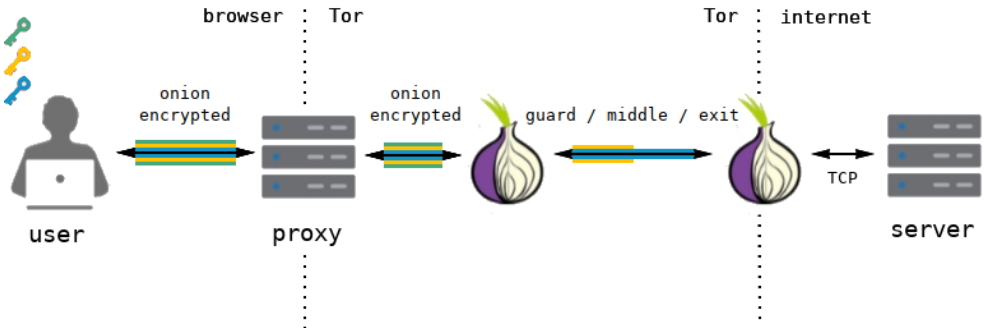


Figure 7: Interaction of Lightnion client running in the user's browser and the Tor network via the Lightnion proxy.

The Lightnion Javascript library authenticates the nodes on the circuit, deriving the correct encryption keys for the symmetric encryption channel, and handles all the necessary cryptographic operations. The proxy simply relays traffic to the corresponding Tor nodes.

The Lightnion Javascript library is provided by the trusted website that the user visits. This website could for example be a secure webmail provider, the trusted medical platform, a collaborative editing platform, or an anonymous questionnaire website.

3.3.1 Parties and trust assumptions

A Lightnion system consists of the following parties.

Service Provider The website that the user visits. The website is trusted and provides the Lightnion Javascript library to the user. Trusting the service provider is natural. The user visits the website, and the user's browser executes any code provided by the service provider. The service provider could always ship defected code, or exfiltrate data using other mechanisms.

Lightnion proxy Untrusted party that proxies the communication between the user's browser and the Tor network. This party is trusted for availability but not for privacy. The Lightnion proxy could even be c-located with a Tor entry node.

End point Users use Lightnion to anonymous contact one of several end points. These end points are untrusted and may try to identify users.

A non-collusion assumption. We assume that the Lightnion proxy does not collude with the end point. If they collude, they can perform a time-correlation attack to deanonimize users. This assumption is also required by Tor to provide security: Tor assumes that no adversary can observe simultaneously messages entering the Tor network via the guard, and messages leaving the network via the exit node.

3.4 Architecture

In Section 3.2 we identified six steps that a normal Tor client must perform to set up an anonymous communication channel via the Tor network. The Lightnion Javascript client retrieves and validates the consensus and Tor node descriptors,

and selects a path. Then, it requests the Lightnion proxy to open a TLS connection to the guard, the first node in the path. See also Figure 6.

Since the proxy, and not the Lightnion client, opens the TLS connection to the guard, the Lightnion client cannot rely on the TLS layer to authenticate the guard. Therefore, the Lightnion Javascript client runs a regular ntor handshake protocol with the guard to authenticate it and to derive the first shared key. All cryptographic operations are performed in the user's browser. The proxy only forwards traffic.

The client continues by requesting the guard to authenticate the circuit to the middle node, and then requests the middle node to extend the circuit to the exit node. At each step, the Lightnion client authenticates the node using an ntor handshake and derives another symmetric encryption key. At the end, the client has established a layered encrypted channel from the users browser, via the proxy, to the exit node. See Figure 7. Note that the Lightnion proxy only sees encrypted traffic.

Finally, the exit node opens a TCP connection to the destination server.

3.5 Evaluation

To evaluate the ease of use and performance of Lightnion we have developed a research prototype of the client and the proxy. We are currently working on turning this prototype into a robust and stable implementation that can be used in a large variety of settings. We report here on the initial research prototype and its performance.

3.5.1 Implementation

We implemented a research prototype of Lightnion. Implementing a minimal Tor client is not easy. Therefore, initially, we developed a new minimal Tor client in Python. This Python implementation parses and validates the consensus and micro descriptors, builds Tor cells, does ntor handshakes to authenticate Tor nodes and derives encryption keys, and handles communication via Tor circuits. This minimal Python implementation highlighted the different steps and complexities in implementing a lightweight Tor client.

Starting from the minimal Python Tor client we split it into two parts: (1) the Lightnion proxy responsible for creating TLS connections to Tor nodes and relaying traffic from the Javascript client, and (2) the Javascript client responsible for the rest. The Javascript client offers a convenient interface for making anonymous TCP connections. This interface hides the complexities of communicating via the Tor network from the developer. See Figure 1.

Currently, the Lightnion Javascript client does not implement all functionality. The Javascript client already supports authenticating Tor nodes, constructing circuits and sending anonymous traffic but the code to validate the consensus and select a path is still being integrated. See Table 1.

3.5.2 Performance

We performed a simple performance evaluation of the current implementation of the Javascript Lightnion library in conjunction with the Python proxy. For these

Table 1: Overview of current implementation status of the Lightnion client. The components marked with a 'P' are implemented as prototypes, but they have not yet been fully merged into the Javascript Lightnion client.

Feature	Status
Parsing and validating consensus	P
Path Selection	P
Authenticate Tor nodes	✓
Constructing a circuit	✓
Communicating via circuit	✓

```
// create a channel through the proxy
lln.open('proxy.example.net', 4990, function(channel)
{
    // callback-oriented interface (skip intermediate states)
    if (lln.state != lln.state.success)
        return

    // create a TCP connection to api.ipify.org on port 80
    tcp = lln.stream.tcp(channel, 'api.ipify.org', 80, handler)
    tcp.send('GET / HTTP/1.1\r\nHost: api.ipify.org\r\n\r\n')
})
```

Listing 1: Sample usage of the Lightnion browser script to make an HTTP request to an external service.

tests we used a small test Tor network constructed using chutney[6]. This network runs 8 Tor nodes (or onion routers, containing at least 2 guards and 5 exit nodes). All tests were performed locally, they therefore do not take network latency into account.

First, we measured the round-trip latency of our implementation using a simple echo server. We created a simple Javascript application running in a browser that uses Lightnion to send packages to this echo service. We measure the time between transmitting the package and when it returns to the Javascript application. We compare the performance of this setup with the baseline of a simple client sending packages using the regular Tor client. All clients, Tor nodes and the echo server run on the same machine. Hence, these experiments measure computational overhead only.

We send packages of 498 bytes (so that they fit within a single Tor cell) at a rate of 250 packages per second. We noticed that the proxy suffers from congestion. Therefore we compare the effect of sending packages for a longer time: 50 package in total (i.e., 200 miliseconds), 500 packages in total (i.e. 2 seconds), and 2500 packages in total (i.e., 10 seconds). See Table 2.

The table shows that longer connections have no significant effect on the latency for clients sent using the regular Tor client. However, whereas the Lightnion Javascript client initially keeps pace with the regular Tor client, sending more packages causes a significant increase in latency. We suspect that mismanaged buffers in the Lightnion proxy are the cause of this. We aim to improve the proxy in the future.

We also profiled our Javascript implementation. About 40% of the time is spent en-

[6] https://gitweb.torproject.org/chutney.git/tree/README

Table 2: Experimental results of latency sending packages to an echo server. We sent packages at a rate of 250 per second. We compare the baseline implementation featuring the regular Tor client to the in browser Lightnion Javascript client. We also measure the length of the total experiment.

	Round-trip Time		Total Time	
#packets	baseline	in browser	baseline	in browser
50	101±18	135±30	200±0	376±10
500	119±77	452±462	2000±0	2493±277
2500	133±94	1397±835	10000±0	11353±522

crypting and decrypting traffic, about 5% of the time is spent performing integrity checks, about 20% is spent on garbage collection, and about 19% is spent on converting between different low-level Javascript data structures. These results show that we could further improve the Javascript library to more efficiently handle garbage collection and type conversion.

3.5.3 Next steps

The next steps for the implementation are as follows:

- Integrate the consensus parsing and validation into the Lightnion Javascript client.
- Integrate the path selection code into the Lightnion Javascript client.
- Restructure the proxy to better deal with multiple current clients and higher rate connections.

As soon as we have completed the above steps to create a fully functioning prototype, we will publish the code for the proxy and the Javascript library on `https://github.com/epfl-spring/` under an open source license.

3.6 Conclusions and future work

In this section we have introduced Lightnion, a module to enable seamless anonyous communication from a user's browsers. The use of Lightnion is invisible users and requires no active involvement from them. We show the feasibility of Lightnion using a research prototype that confirms that it is easy to use and can perform well for many scenarios. We aim to turn the research prototype into a well-performing piece of software that can be used in a large number of scenarios.

4 Claimchain

In order to encrypt messages, both MLS and Autocrypt require knowledge of the keys of communication partners. In Deliverable 2.2 we presented Claimchains, that support a federated identity system in which users can attest about the validity of other users' keys in a privacy-friendly manner. While integrating Claimchains into Autocrypt we discovered that malicious users could abuse Claimchain's privacy-friendly sharing mechanisms to lie about the keys they gossip. We now present a

new version of Claimchain that prevents this attack. We formally prove its security and empirically show that the new additions have a negligible impact on performance.

4.1 Introduction

A ClaimChain, as described in D2.2, is a cryptographic construction for privacy-preserving authentication of public keys that can enable secure and privacy-friendly decentralized key distribution. Users store claims about their own keys and their believes about other people's keys in the ClaimChain data structure. These chains form authenticated decentralized repositories that enable users to prove the authenticity of both their keys and the keys of their contacts. ClaimChains are encrypted, and therefore protect the stored information, such as keys and contact identities, from prying eyes. At the same time, it contains mechanisms to reliably and efficiently prevent that a user can lie about identity-key bindings within a block.

ClaimChain's decentralized approach alleviates the privacy problem of centralized certification authorities, such as SKS Keyservers[7] for PGP keys, which can observe users' key look-ups, and thus can infer their communication patterns. However, since no one has a global view of all the bindings in Autocrypt's decentralized approach, malicious users or providers can supply different user-to-key bindings to different recipients, effectively opening the door to man-in-the-middle attacks. ClaimChains includes mechanisms to reliably and efficiently prevent equivocation within a block.

While integrating ideas from ClaimChains into Autocrypt version 2, we discovered a new attack that uses the privacy-enabling access control mechanism to allow ClaimChain owners to equivocate *across* blocks. We adjusted the ClaimChain data structures and cryptography to allow chain owners to prove that they did not equivocate without needing to reveal any other private information. Using this proof, recipients of claims can verify that the claim owner did not equivocate across blocks.

To ensure that the new ClaimChain datastructure is secure and private, we modeled the security and privacy properties of the constructions and then proved its security. These proofs, presented in Appendix A were also not present in deliverable "D2.2: Federated Secure Identity Protocol".

Changes with respect to deliverable D2.2. In deliverable "D2.2: Federated Secure Identity Protocol" we presented an earlier version of ClaimChains. In this deliverable we present an updated version of ClaimChain. This new version differs from the original in the following aspects:

1. We changed the cryptographic data structures to allow chain owners to prove that they did not equivocate between blocks without revealing any privacy-sensitive information.
2. We show how to construct proofs of non inter-block equivocation, see Section 4.4.1
3. We formally model and proof the security and privacy properties of the new ClaimChain data structure. See Appendix A.

Allowing the proofs of non inter-block equivocation required many small changes to the data structures. Therefore, we are presenting the entire construction rather

[7] `https://sks-keyservers.net`

than focusing on the changes only.

Published at a scientific conference. The improved version of ClaimChain that we present here has also been published at the Workshop on Privacy in the Electronic Society:

> Bogdan Kulynych et al. "ClaimChain: Improving the Security and Privacy of In-band Key Distribution for Messaging". In: *WPES 2018*. 2018, pp. 86–103. DOI: 10.1145/3267323.3268947

4.2 Problem statement and goals

We assume a messaging system in which users embed their cryptographic keys in-band, i.e., into the messages themselves or into the message headers, as in Autocrypt. These keys are used to provide message confidentiality using opportunistic encryption [Duk14]. That is, the communication is encrypted when users know each others' keys, but falls back to plaintext when they do not.

Sending keys as part of message headers results in two problems. First, in terms of privacy, adding such headers reveals users' social ties. Second, in terms of security, man-in-the-middle attackers can modify the header contents, since they are not authenticated. Moreover, malicious users can equivocate about others' keys.

Design goals. We assume that all actors in the system, users and providers, may act maliciously. Our goal is to design a data structure that can store the binding between keys and identities, and is suitable for integrating with in-band key distribution. The purpose of the structure is to support key validation, i.e., help users establish the authenticity of user-key bindings, as long as some users in the system are honest. Furthermore, it must protect users' privacy without relying on centralized parties.

More concretely, we aim at providing the following properties. First, the structure must guarantee the *integrity* and *authenticity* of identity-key bindings, i.e., it should not be possible to replace or inject bindings without being detected. Second, we want to preserve the *privacy of cross-referenced information* and the *privacy of the social graph*. These properties ensure that only authorized users can access the key material in the structure and the identities of the bindings being distributed. Third, the structure must prevent users from *equivocating* other users with respect to the identity-key bindings that they share. That is, a user Owen should *not* be able to show to Alice and Bob different versions of a Charlie's key, even if he withdraws Alice's access to see Charlie's keys. In the latter case if Alice ever regains access, she must be able to detect Owen's misbehavior. Finally, our construction should not entail significant computational or communication overhead for the end users and providers to enable adoption at large scale.

Non-goals. In-band key distribution cannot ensure full availability of public encryption keys. The keys of one or more recipients may not be available to a sender at a time of sending a message, and thus, because of the opportunistic encryption operation, the message would be sent in the clear. We are therefore not concerned with ensuring 100% availability of keys. Instead, our goal is to secure the keys that *are* distributed without harming privacy. If guaranteeing encrypted communication is absolutely necessary, parties must exchange keys in a reliable way, e.g. through a centralized service or an out-of-band mechanism.

Furthermore, throughout this paper we consider that users have only one identity,

and use one and only one structure to store key bindings of their contacts. If a user wishes to have different identities, she must create one structure per identity.

4.3 ClaimChain design

In this section we introduce ClaimChain, a structure to store key bindings in a secure and privacy-friendly manner.

4.3.1 Cryptographic preliminaries and notation

We denote sampling uniformly at random from a set X as $x \leftarrow\$ X$, and the assignment of an evaluation of a function $f(x)$ to y as $y \leftarrow f(x)$, regardless of whether f is probabilistic or deterministic. We denote concatenation of strings by $\|$.

Let λ be the security parameter. ClaimChain relies on the following standard cryptographic primitives. Let $\mathsf{Enc}(k, m) \mapsto c$ and $\mathsf{Dec}(k, c)$ denote an IND-CPA secure symmetric authenticated encryption scheme. ClaimChain uses an existentially unforgeable signature given by the algorithms $\mathsf{Sig.KeyGen}(1^\lambda)$ returning the keypair $(\mathsf{sk}_{\mathsf{sig}}, \mathsf{pk}_{\mathsf{sig}})$, $\mathsf{Sign}(\mathsf{sk}_{\mathsf{sig}}, m)$ returning a signature σ, and the verification function $\mathsf{Sig.Verify}(\mathsf{pk}_{\mathsf{sig}}, \sigma, m) \mapsto \{\top, \bot\}$. We write $\mathsf{DH.KeyGen}(1^\lambda)$ for the generation of a Diffie-Hellman (DH) keypair $(\mathsf{sk}_{\mathsf{DH}}, \mathsf{pk}_{\mathsf{DH}})$ using which we can non-interatively compute the shared DH key $s \in \{0,1\}^*$ using $\mathsf{SharedSecret}(\mathsf{sk}_{\mathsf{DH}}, \mathsf{pk}^R_{\mathsf{DH}})$. Finally, let H be a cryptographic hash function from which we derive a family of hash functions $H_i : \{0,1\}^* \rightarrow \{0,1\}^{2\lambda}, i > 0$.

All schemes use a cyclic group $\mathbb{G}$ of prime order q generated by g. We write $\mathbb{Z}_q$ for the integers modulo q. Moreover, we assume the existence of a cryptographic hash function $H_{\mathbb{G}} : \{0,1\}^* \rightarrow \mathbb{G}$ that hashes strings to group elements, and a hash function $H_q : \{0,1\}^* \rightarrow \mathbb{Z}_q$ that hashes strings to the elements of $\mathbb{Z}_q$.

ClaimChains also require an information-theoretically hiding commitment scheme $\mathsf{Commit}(r, m)$ that commits to values $m \in \mathbb{Z}_q$ given a randomizer $r \in \mathbb{Z}_q$. We instantiate this scheme using Pedersen's commitment scheme [Ped91]. Let $g_1, g_2 \in \mathbb{G}$ be random generators such that the discrete logarithms of g_1 and g_2 with respect to each other are unknown. Then, $\mathsf{Commit}(r, m) = g_1^r g_2^m$.

ClaimChains use standard zero-knowledge proofs of knowledge, and in particular Schnorr's proof of knowledge of discrete logarithms [Sch91], to prove correctness of claims. We use the Fiat-Shamir heuristic [FS86] to derive non-interactive signature proofs of knowledge. For example, we write:

$$\mathsf{SPK}\left\{(r, m) : C = g_1^r g_2^m\right\}(t)$$

to denote the non-interactive signature proof of knowledge on a random string t for which the prover knows the commitment opening (r, m). To focus on the semantics of the proof, we write

$$\mathsf{SPK}\left\{(r, m) : C = \mathsf{Commit}(r, m)\right\}(t)$$

instead, to denote the same proof.

Finally, ClaimChains use a verifiable random function (VRF) [MRV99; FZ13], given by the algorithms VRF.KeyGen and VRF.Eval. The function $\mathsf{VRF.KeyGen}(1^\lambda)$ returns a keypair $(\mathsf{sk}_{\mathsf{VRF}}, \mathsf{pk}_{\mathsf{VRF}}) = (\mathsf{sk}_{\mathsf{VRF}}, g^{\mathsf{sk}_{\mathsf{VRF}}})$. Then, $h = \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, m) =$

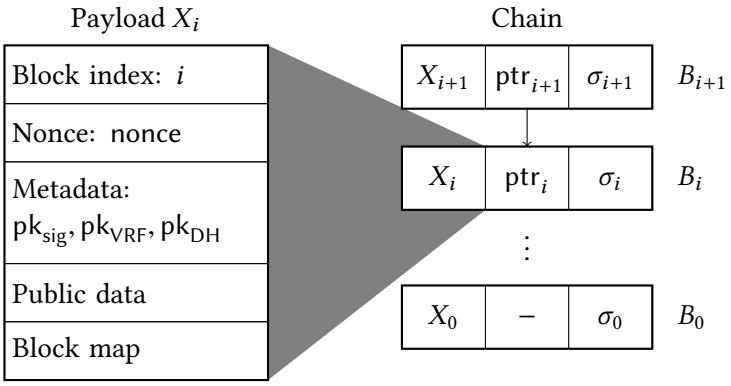


Figure 8: ClaimChain block structure

$H_{\mathbb{G}}(m)^{\mathsf{sk}_{\mathsf{VRF}}}$ is the VRF of the value m. Users prove the that h was correctly computed by constructing the proof

$$\mathsf{SPK}\left\{(\mathsf{sk}_{\mathsf{VRF}}) : \mathsf{pk}_{\mathsf{VRF}} = g^{\mathsf{sk}_{\mathsf{VRF}}} \wedge h = \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, m)\right\}().$$

The properties of VRF hashes are similar to that of cryptographic hashes: uniqueness of h for a given message and private key, collision resistance, and pseudorandomness (assuming no access to the corresponding proof) [Pap+17].

4.3.2 Overview

We consider that each user has a state made of information about herself and her beliefs about other users' states. At a given point in time a users' state is represented as a set of statements, called *claims*. Claims can be of two kinds. The first type of claim refers to a user's own state. In particular, these may be statements on the user's encryption keys, identity information (screen name, real name, or e-mail), or other cryptographic material such as verification keys to support digital signatures. The second type of claims, we call them cross-references, refer to other users' states. A claim owner creates a cross-reference to endorse the referenced user's state as being authoritative, i.e., a cross-reference indicates the owner's belief that the self key material found in those users' state is correct. A user's state evolves over time as she rotates her keys and observes the evolution of others' states. She stores snapshots of her state in a cryptographic data structure called a *ClaimChain*.

The core element of a ClaimChain is a block. A block includes all claims that the owner endorses at the time when she creates the block, i.e., a block is a snapshot of the owner's state. Blocks form a chain. A block contains a payload X, a pointer to the previous block ptr, and a digital signature σ_i on the payload and the pointer. See Figure 8. The payload of the previous block X_{i-1} contains the verification key $\mathsf{pk}_{\mathsf{sig}}^{(i-1)}$ for the private key $\mathsf{sk}_{\mathsf{sig}}^{(i-1)}$ that signs σ_i.

We now describe each of the block components in detail.

The payload X_i has the following content (see Figure 8, left):

- *Block index.* The block's position in the chain. The index of the genesis block is 0.
- *Nonce.* A fresh cryptographic nonce used to 'salt' all cryptographic operations within the block. It ensures that the information across blocks is not linkable.

- *Metadata.* The current signature verification key of the owner $\mathsf{pk}_{\mathsf{sig}}$, that is used to authenticate the next block of the ClaimChain; the current key $\mathsf{pk}_{\mathsf{VRF}}$ to compute a verifiable random function used to support non-equivocation; and a Diffie-Hellman key $\mathsf{pk}_{\mathsf{DH}}$ used to provide claim privacy.
- *Public data.* Application-specific data the owner wishes to make publicly visible. For in-band key distribution we set this to the owner's self-claim on her current public encryption key.
- *Block map.* A high-integrity key-value map storing the claims, as well as access tokens that express access-control rights. This map has two core properties: i) a key can only be resolved to a single value, and ii) it enables the generation and verification of efficient proofs of inclusion of claims or access tokens. We implement the map using *unique-resolution key-value Merkle trees*, explained in more detail in Section 4.3.3. For our use case the map only contains cross-references.

The signature $\sigma_i = \mathsf{Sign}(\mathsf{sk}_{\mathsf{sig}}^{(i-1)}, (X_i, \mathsf{ptr}_i))$ authenticates the current block. A block B_i must have a valid signature under the verification key indicated in the payload of the previous block B_{i-1}. The genesis block of a ClaimChain is 'self-signed'. The corresponding initial public signing key is included in the initial payload. Each block in the chain contains enough information to authenticate past blocks as being part of the chain, validate the next block, and, by transitivity, all future blocks as being valid updates. Therefore, a user with access to a block of a chain that she believes is authoritative, can both audit past states of the chain, and authenticate the validity of newer blocks.

4.3.3 Low-level operations

We now describe how we implement claims and access tokens, and how they are combined into the block map.

Claims. We model claims as a tuple composed of a *label* l and a *body* m. The label is a well-known identifier associated with the identity of the user to whom the claim refers. The body is the state of that user at the time when the claim is generated, represented as the latest block of this user's ClaimChain. For instance, a claim ('bob@gmail.com', B) represents the ClaimChain owner's belief that the current state of the user associated with this Gmail account is represented by the block B.

For privacy reasons, claims in a ClaimChain are encrypted. Thus, they cannot be found directly by other users. To enable efficient search for concrete claims within a ClaimChain block, we introduce a *lookup key*, or *index*, i for each claim.

We illustrate the encoding of claims in procedure EncClaim, see Figure 9. Consider a claim (l, m) that is to be included in a block. We first compute the unique VRF of its label and derive the claim's lookup key i [lines 2–3]. Note that the computation of h includes a per-block nonce to ensure that the lookup keys for a given claim label look different across blocks, and therefore no patterns can be inferred from their appearance.

Recall that VRF hashes are unique. We use them to derive lookup keys to ensure that, given a label, all users retrieve the same claim, effectively supporting non-equivocation within a block. This use of VRFs is inspired by CONIKS [Mel+15]. We

procedure EncClaim($\mathsf{sk}_{\mathsf{VRF}}, l, m, \text{nonce}$)
  $h \leftarrow \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, l \parallel \text{nonce})$
  $i \leftarrow H_1(h)$
  $r \leftarrow\$\ \mathbb{Z}_q, \quad \mathsf{com} \leftarrow \mathsf{Commit}(r, H_q(m))$
  $k_\pi \leftarrow\$\ \{0,1\}^\lambda$
  $\pi \leftarrow \mathsf{SPK}\{(\mathsf{sk}_{\mathsf{VRF}}, r) : \mathsf{pk}_{\mathsf{VRF}} = g^{\mathsf{sk}_{\mathsf{VRF}}} \wedge$
    $h = \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, l \parallel \text{nonce}) \wedge$
    $\mathsf{com} = \mathsf{Commit}(r, H_q(m))\}(k_\pi)$
  $k \leftarrow\$\ \{0,1\}^\lambda, \quad c \leftarrow \mathsf{Enc}(k, \pi \parallel m) \parallel \mathsf{com}$
  return $r, h, k, k_\pi, (i, c)$

procedure EncCap($\mathsf{sk}_{\mathsf{DH}}, \mathsf{pk}^R_{\mathsf{DH}}, l, h, k, k_\pi, \text{nonce}$)
  $s \leftarrow \mathsf{SharedSecret}(\mathsf{sk}_{\mathsf{DH}}, \mathsf{pk}^R_{\mathsf{DH}})$
  $i_{\mathsf{cap}} \leftarrow H_3(s \parallel l \parallel \text{nonce})$
  $k_{\mathsf{cap}} \leftarrow H_4(s \parallel l \parallel \text{nonce})$
  $\mathsf{cap} \leftarrow \mathsf{Enc}(k_{\mathsf{cap}}, h \parallel k \parallel k_\pi)$
  return $(i_{\mathsf{cap}}, \mathsf{cap})$

procedure DecClaim($\mathsf{pk}^O_{\mathsf{VRF}}, h, l, k, k_\pi, c, \text{nonce}$)
  $\bar{c} \parallel \mathsf{com} \leftarrow c$
  $\pi \parallel m \leftarrow \mathsf{Dec}(k, \bar{c})$
  ▷ Note the verification of π requires $\mathsf{pk}^O_{\mathsf{VRF}}$, h, l, k_π, com, m, and nonce.
  if π is not a valid proof **then**
    return $\perp$
  return m

procedure DecCap($\mathsf{sk}_{\mathsf{DH}}, \mathsf{pk}^O_{\mathsf{DH}}, l, \mathsf{cap}, \text{nonce}$)
  $s \leftarrow \mathsf{SharedSecret}(\mathsf{sk}_{\mathsf{DH}}, \mathsf{pk}^O_{\mathsf{DH}})$
  $k_{\mathsf{cap}} \leftarrow H_4(s \parallel l \parallel \text{nonce})$
  $h \parallel k \parallel k_\pi \leftarrow \mathsf{Dec}(k_{\mathsf{cap}}, \mathsf{cap})$
  $i \leftarrow H_1(h)$
  return i, h, k, k_π

Figure 9: Low-level ClaimChain operations

also include additional cryptographic elements in our encoded claims in order to obtain a stronger non-equivocation property than CONIKS. Specifically, we guarantee that equivocation is detectable *across blocks* without the need for key owners to intervene. The need for a detection mechanism stems from the fact that ClaimChain owners can give and withdraw access to claims at will. Thus, they can try to equivocate others by giving them access to different information in different blocks. To make this misbehaviour detectable we provide ClaimChain owners with the ability to prove statements about claims that other users cannot see. This way, if a proof cannot be completed, equivocation is revealed (see Section 4.4.1 for more details).

To prove statements on claim contents without revealing them, we commit to the claim body m [line 4]. Moreover, we construct a non-interactive proof π on k_π proving that the VRF h is correct and that the commitment com commits to m [lines 5–6]. When decoding a claim, users verify the proof π. The proof verification key k_π ensures that only authorized users can verify this proof.

Once π is computed, we encrypt m and π with a random key k [line 7]. Finally, the claim encoding consists of this ciphertext and the commitment com: $c = \mathsf{Enc}(k, \pi \parallel m) \parallel \mathsf{com}$.

The binding property of the commitment com and the validation provided by the proof π also ensure that all users with access to an encoded claim c must recover the same claim body m. This makes this encoding scheme an instance of committing, or non-deniable, encryption [GLR17]. Hence, a malicious owner can not equivocate by supplying two different claim encryption keys to different users.

The procedure DecClaim, see Figure 9, describes the decoding of a claim. It takes as input the encryption key k, the VRF hash h, and the proof verification key k_π from the owner (see below). Then, users can decrypt the ciphertext using k [lines 2–3]; and verify the claim proof π, which includes the verification of the correctness of the VRF hash h and of the commitment com [lines 4–6].

Our claim encoding scheme offers four distinct security advantages. First, the use of the VRF ensures that lookup keys can only be produced by the owner of the chain, which as we describe below supports access control. Second, the lookup key is unique for a given label, and thus can be used to support non-equivocation for claims within a block. Third, the lookup key i and claim encoding c leak no information about the claim label or body. Fourth, it supports zero-knowledge proofs about claim contents, which enables the detection of equivocation across blocks.

Access capabilities. ClaimChain owners create cryptographic access tokens called *capabilities* to ensure that only authorized users can access specific claims. A single capability grants one authorized user access to one claim. We call the authorized users *readers*.

An encoded capability is an encryption of all the values needed to obtain a claim lookup key and decode the corresponding claim: the encryption key k, the VRF hash h, and the proof verification key k_π. We encrypt these using a key derived from a shared secret s between the chain owner and the reader. Similarly to claims, encoded capabilities have an associated lookup key i_{cap}, and a body cap.

The procedure EncCap, see Figure 9, describes how to encode capabilities. First, it computes the shared Diffie-Hellman secret s using the owner's private DH key $\mathsf{sk}_{\mathsf{DH}}$ and reader's public DH key $\mathsf{pk}^R_{\mathsf{DH}}$ [line 2]. The latter is available in the metadata of the reader's ClaimChain. We use the secret s to derive both the capability lookup key i_{cap} [line 3], and the capability encryption key k_{cap} [line 4]. Then we encrypt

the values h, k, and k_π using the key k_{cap} to obtain the capability encoding [line 5]: $\text{cap} = \text{Enc}(k_{\text{cap}}, h \parallel k \parallel k_\pi)$.

Chain owners store the encoded claim c under the lookup key i in the block map. Similarly, they store the encoded capability cap under the lookup key i_{cap}. To find a capability corresponding to a claim with label l in a ClaimChain block, a reader first computes the lookup key i_{cap} for label l using the shared secret with the ClaimChain owner. If the corresponding capability cap is in the block, she decodes it using DecCap, see Figure 9. First, the reader derives the shared secret s [line 2], and computes the capability encryption key k_{cap} using the claim label l [line 3]. She can then decrypt cap using k_{cap} [line 4], obtaining the label's VRF hash h, the encryption key k, and the proof verification key k_π. With this information the reader can compute the claim's lookup key $i = H_1(h)$, find the claim, and decode it using DecClaim.

Block map. Encoded claims and capabilities are stored in the block map. We implement the block map using a unique-resolution key-value Merkle tree. Unlike a standard Merkle tree that implements an authenticated set data structure, a key-value tree is an instance of an authenticated dictionary [CW11]. It can be efficiently queried for a value that corresponds to a given lookup key. Our construction is similar to that of a binary search tree: the intermediate nodes contain pivots that define whether the querier should follow the left child or a right child; the leaf nodes contain the values. The construction allows queriers to be sure that retrieved values are unique, i.e., there cannot exist any other leaf nodes that correspond to the queried lookup key. We call this the *unique resolution property*. We formally define the property in Experiment 1 and prove it in Theorem 3 (both in Appendix A.1). We refer to Appendix A.1 for further details on the construction.

The unique-resolution property guarantees that for a given lookup key i, respectively i_{cap}, there can only be one claim c, respectively capability cap. The uniqueness of the VRF value h, the property of the tree, and the commitment in the claim encoding, ensures that a ClaimChain owner can not equivocate within a block.

We note that ClaimChain blocks only need to include the root hash of the Merkle tree, not the whole tree. This is because our Merkle tree construction allows to produce an inclusion proof for items: a path from the root to the leaf node which contains the item. Thus, providing others with this paths is enough to convince them that the items are in the tree defined by the root in the block.

4.3.4 High-level operations

So far we have described how users can encode and decode claims. We now outline how these claims can be included in a ClaimChain and read from it. At a glance, owners create blocks with a set of encoded claims and corresponding encoded capabilities, and use them to extend their ClaimChains. Any user can validate the authenticity and integrity of the chain. Moreover, readers can retrieve the claims they are authorized to read. Section 4.4 illustrates how these operations can be used in the context of in-band key distribution.

Content-addressable store. ClaimChain owners store their blocks and trees in mutable *content-addressable stores*. These are key-value stores where the key must be the hash of the corresponding value. They are a good fit for ClaimChains because i) it is easy to verify their integrity by checking that all keys are the hashes of the respective objects they map to; and ii) an incomplete store cannot lead to an

erroneous decision on the authenticity, inclusion or exclusion of any block or tree node. The store supports two operations:

- PUT(v). Record the value v in the store.
- GET(h). Return v such that $h = H(v)$, if present in the store.

Extending a chain. Whenever an owner decides to add new claims to her ClaimChain she uses the procedure EXTENDCHAIN in Figure 10. This procedure takes as input the public application data, a set of claims (l_j, m_j) to add to the block, an access control set acs consisting of the authorized reader-label pairs for these claims, the cryptographic keys necessary to create the block (keypairs for signatures, DH key exchange, and VRF), as well as the previous signing key $\mathsf{sk}'_{\mathsf{sig}}$ included in the previous block, the pointer ptr to that block, and, finally, the user's store.

To create a block the user first generates a random nonce that is used for all encoding operations [step 1]. She then encodes all the claims and capabilities [steps 2–3]. The set S of encoded values and their respective lookup keys are used to construct a Merkle tree with root hash MTR, as described in Algorithm 1 in Appendix A.1 [steps 4–5]. She then constructs the block payload X using the nonce, the block metadata containing the public keys $(\mathsf{pk}_{\mathsf{DH}}, \mathsf{pk}_{\mathsf{sig}}, \mathsf{pk}_{\mathsf{VRF}})$, the public application data, and the root MTR of the Merkle tree. She signs the payload X and the pointer ptr to the previous block using the previous signing key $\mathsf{sk}'_{\mathsf{sig}}$ (see Figure 8) [step 6]. Finally, she puts the obtained block, $B = (X, \mathsf{ptr}, \sigma)$, into the content-addressable store [step 8].

Chain validation. Readers must always validate that new blocks correctly extend the chain that they have previously seen. To do so, users run the procedure VALIDATEBLOCKS, see Figure 11. The input to this procedure is a list of blocks B_i, where B_0 is the last validated block, and B_1 through B_t are the new blocks to be validated. For each new block B_i the reader first checks if the block includes all elements: the payload, signature, and the pointer [step 1]. Next, she retrieves the public key $\mathsf{pk}_{\mathsf{sig}}$ from the preceding block B_{i-1} and verifies the signature in the block B_i [steps 2–3]. This verifies the authenticity of the chain. Finally, she verifies that the pointer in the block B_i is a hash of the preceding block B_{i-1}, which verifies the integrity of the chain [step 4].

Retrieval of the claim by label. After having validated the ClaimChain of an owner, the reader can query it to retrieve claims of interest using procedure GETCLAIM in Figure 10. This procedure takes as input the reader's private Diffie-Hellman key $\mathsf{sk}_{\mathsf{DH}}$, the claim label l, a pointer to the latest block ptr and the owner's store. The reader retrieves the block, and parses it to get the block's nonce, the owner's public keys, and the block map hash [steps 1–2]. She then derives the capability lookup key using the DH secret shared with the owner [step 3], queries the block map to retrieve the corresponding capability [step 4]. We refer to Algorithm 2 in Appendix A.1 for the details of the QUERYTREE algorithm. Next, she runs the decoding procedure to obtain the claim lookup key i, the VRF hash h, the claim encryption key k, and the proof verification key k_π [steps 5]. She then obtains the claim encoding c by querying the tree with the claim's lookup key i [step 6]. Finally, the reader decodes and verifies the encrypted claim using h, k, k_π [step 7].

procedure ExtendChain(data, claims, acs, keys, ptr, store)

1. Randomly generate a λ-bit nonce nonce.
2. For each claim (l, m) in claims:

$$r, h, k, k_\pi, (i, c) \leftarrow \text{EncClaim}(\text{sk}_{\text{VRF}}, l, m, \text{nonce}),$$

   Additionally, record each randomizer r.
3. For each tuple $(\text{pk}^R_{\text{DH}}, l)$ in acs, encode a capability:

$$(i_{\text{cap}}, \text{cap}) \leftarrow \text{EncCap}(\text{sk}_{\text{DH}}, \text{pk}^R_{\text{DH}}, l, h, k, k_\pi, \text{nonce})$$

4. Construct a set S containing the encoded claims and capabilities.
5. Build a unique-resolution key-value Merkle tree from the set of entries S: $\text{MTR} \leftarrow \text{BuildTree}(S, \text{store})$.
6. Compute the block

$$B \leftarrow \Big((X, \text{ptr}), \sigma = \text{Sign}(\text{sk}'_{\text{sig}}, (X, \text{ptr}))\Big)$$

   where X contains the nonce, MTR, the block metadata (containing the public keys $\text{pk}_{\text{DH}}, \text{pk}_{\text{sig}}, \text{pk}_{\text{VRF}}$), and the public application data.
7. Put the block B into the store using $\text{Put}(B)$
8. **return** $H(B)$.

procedure GetClaim(sk_{DH}, l, ptr, store)

1. Get the block from the store: $B \leftarrow \text{Get}(\text{ptr})$.
2. Retrieve owner's public keys $(\text{pk}^O_{\text{DH}}, \text{pk}^O_{\text{VRF}}, \text{pk}^O_{\text{sig}})$, the block's nonce nonce, and the block map hash MTR from block B.
3. Compute the capability lookup key:

$$i_{\text{cap}} \leftarrow H_3(s \parallel l \parallel \text{nonce}),$$

   where s is the shared secret $s = \text{SharedSecret}(\text{sk}_{\text{DH}}, \text{pk}^O_{\text{DH}})$.
4. Get the encoded capability from the tree:

$$\text{cap} \leftarrow \text{QueryTree}(\text{MTR}, i_{\text{cap}}, \text{store}),$$

5. Obtain the claim lookup key i, the VRF hash h, the claim encryption key k, and the proof verification key k_π:

$$i, h, k, k_\pi \leftarrow \text{DecCap}(\text{sk}_{\text{DH}}, \text{pk}^O_{\text{DH}}, l, \text{cap}, \text{nonce})$$

6. Get the encoded claim from the tree:

$$c \leftarrow \text{QueryTree}(\text{MTR}, i, \text{store})$$

7. Decode c and verify the correctness of the claim:

$$m \leftarrow \text{DecClaim}(\text{pk}^O_{\text{VRF}}, l, h, k, k_\pi, c, \text{nonce})$$

8. If any of the lookups failed, return None. If the verification failed, return $\perp$. Otherwise, return m.

Figure 10: Extending and querying ClaimChains

procedure ValidateBlocks($\{B_0, B_1, B_2, ..., B_t\}$)

For each i from 1 to t:

1. Check that block B_i is of the form $((X_i, \mathsf{ptr}_i), \sigma_i)$.
2. Retrieve the public key $\mathsf{pk}_{\mathsf{sig}}^{(i-1)}$ from the block B_{i-1}.
3. Verify B_i's signature σ_i using the previous block's key:

$$\mathsf{Sig.Verify}(\mathsf{pk}_{\mathsf{sig}}^{(i-1)}, \sigma_i, (X_i, \mathsf{ptr}_i))$$

4. Verify that the hash chain construction is correct:

$$\mathsf{ptr}_i = H(B_{i-1})$$

Return $\top$ if all checks were successful, otherwise return $\bot$.

Figure 11: Block validation

4.3.5 Security and privacy properties

We now sketch why the ClaimChain design fulfills the security and privacy objectives established in Section 4.2.

We note that *authenticity* and *integrity* are guaranteed through the usage of signature and hash chains respectively. Signatures guarantee that the information stored in a ClaimChain has been added by the owner of the chain. The usage of cryptographic hash functions for constructing the pointers between blocks guarantees that tampering with the ClaimChain content will be detected.

Privacy. ClaimChains provide *privacy of content* and *privacy of the social graph*. We capture these through the following properties:

- *Capability-reader unlinkability.* The adversary cannot determine for which honest user a capability has been created.
- *Claim privacy.* The adversary cannot learn anything about the labels and bodies of claims for which it does not have the corresponding capabilities.

Informally, these properties are provided by ClaimChains because the adversary can neither derive the capability lookup key, nor learn the contents of the encoded capability without the knowledge of the shared secret used to encrypt the them. This implies that an adversary without this key cannot read capabilities nor learn to whom they are destined (capability-reader unlinkability). Since the adversary cannot read the capability, it also does not learn the VRF hash h required to compute the claim lookup key, nor the claim encryption key k. Moreover, the pseudorandomness of the VRF hash h ensures that the adversary cannot compute h without the cooperation of the chain owner. Thus, the adversary cannot check whether a particular claim is included in the block.

Following a similar reasoning, the adversary cannot learn the content of a claim from its lookup key. Furthermore, the encoded claim c does not reveal anything about the claim, except its length. Therefore, claim privacy holds, as long as all claims are of the same length, or padded to the same length.

We formalize these properties in Experiments 2 and 3, and prove them in Theorems 4 and 5 in Appendix A.2.

Non-equivocation. Our construction also *prevents equivocation*. Specifically, it guarantees the following two properties:

- *Intra-block non-equivocation.* Within a given block, a ClaimChain owner cannot include two different bodies encrypted to different readers, having the same claim label.
- *Detectable inter-block equivocation.* For any subset of ClaimChain blocks the owner can produce a proof that, for a given label l, all claims in these blocks belong to some set of allowed claims M without revealing the claims themselves.

The latter property ensures that a user cannot selectively withdraw access rights between blocks to equivocate users. We detail this attack and the proof that mitigates it in Section 4.4.1.

The intra-block non-equivocation relies on three properties of the ClaimChain construction. First, the uniqueness of the VRF hash h ensures that for a given label all readers will compute the same claim lookup key. Second, the unique-resolution property of our Merkle tree ensures that for a given lookup key all readers obtain the same claim encoding. Third, the claim commitment ensures that all readers will decrypt the same claim body.

We formalize both properties in Experiments 4 and 5, and prove them in Theorems 6 and 7 respectively in Appendix A.2.2.

4.4 Using ClaimChains to secure in-band key distribution

Recall from Section 4.2 that the goal of the ClaimChain data structure is to improve the security and privacy of in-band key distribution. In this section we describe how this can be achieved.

Building a ClaimChain. To use a ClaimChain, a user has to build blocks, containing her claims. When to update the ClaimChain depends on the owner's preferences. For example, a user can update her chain whenever she rotates her own encryption public key, or when she needs to distribute new cross-references that are not present in her ClaimChain yet.

To update her chain, an owner runs the ExtendChain procedure (Figure 10). For this purpose, she encodes a set of claims representing all her current views of other users as cross-references in the following way. For each contact, she makes a cross-claim (l, m), where l is the contacts' e-mail, and m is the contact's latest block.

Then, the owner must decide which of these claims she intends to make available to which of her contacts. This choice determines the access control set acs. The access control policy is governed by the user's privacy preferences. Defining these preferences is beyond the scope of this work.

Recall that to implement access control the owner uses shared DH secrets with each of the readers. Thus, the owner needs to complete a round-trip of messages with a contact before she can give this contact access to her claims.

Finally, the owner puts her own public encryption key into the public application data section of the block. For our use case of in-band key distribution we assume that all keys are constant size. Hence blocks, and therefore claims, are constant size

too. This ensures claim privacy even though the encryption scheme leaks the length of the plaintext.

Distributing ClaimChains. To fulfill their purpose, ClaimChains must be made available to other users. For this, a user includes a content-addressable store containing blocks from her ClaimChain, and a subset of the Merkle tree nodes from her latest ClaimChain block, in every message she sends. The user keeps a record of which blocks they have sent to whom. To select the blocks to be sent, the sender checks her record, and includes all her ClaimChain blocks that the recipients of the current e-mail have not received yet.

The subset of the Merkle tree is selected to ensure that all information in the ClaimChain relevant to her message can be authenticated. More concretely, the sender produces resolution paths on the tree (see the GetIncPath procedure in Algorithm 2 in Appendix A.1 for the details) for each relevant claim and capability.

Receiving messages and validating ClaimChains. Upon receiving a message with a store containing ClaimChain data, a user first validates the received chain, running the ValidateBlocks procedure (Figure 11) to check if the new blocks extend a chain that has been seen previously. If the validation succeeds, the owner checks the *consistency* of the cross-references in the newly received part of the chain, i.e., whether all the cross-references to Charlie point to the blocks on a single chain. This partially prevents malicious chain owners from cross-referencing fake chains. See Section 4.4.1 for an example of such an attack, and the details of a consistency check procedure in case the receiver does not have access to the claim in some of the received blocks. If both checks succeed, she stores all the received blocks and tree nodes into her *gossip storage*. This enables her to query the sender's ClaimChain later. The gossip storage contains all the block and tree nodes the user has received over time.

Message encryption. Following the opportunistic encryption paradigm, before sending a message, the sender checks if she has learned the public keys of all of the recipients through the ClaimChains she has received over time. If she cannot find all keys, she sends the message in plaintext.

To find the encryption keys she proceeds as follows. For every recipient with e-mail address l, and every ClaimChain with head ptr in her gossip storage gossip_store, she runs GetClaim($\mathsf{sk}_{\mathsf{DH}}, l$, ptr, gossip_store), see Figure 10, to find out whether it includes cross-references to this recipient. For every hit, she parses the corresponding claim and adds the cross-referenced ClaimChain block of the recipient to a *social evidence set* for this recipient. She then identifies the most recent block of the recipient's ClaimChain (out of those present in her evidence set), i.e., the one that forms the longest hash chain, and uses the encryption public key in that block to encrypt the message. As a result of this process, the sender may discover new blocks of the recipient's chain. She can then include the updated views as cross-references next time the chain is extended.

Resolving conflicts. This key resolution process may reveal conflicting views. For example, the blocks in the evidence set could point to two or more distinct chains. Another possibility is there could be a 'fork': two valid blocks with the same block index that extend a common parent block. In either case, ClaimChains conflicts are detectable and generate cryptographically non-repudiable evidence. The design of mechanisms for sharing such evidence and deciding how to act on it is out of scope of this work.

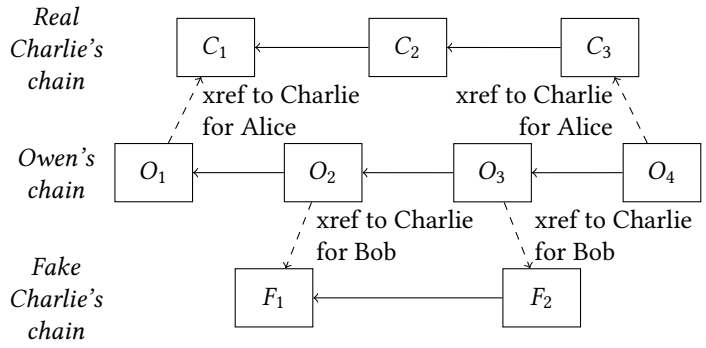


Figure 12: Inter-block equivocation

4.4.1 Detecting inter-block equivocation

ClaimChain's intra-block non-equivocation property ensures that all readers of cross-references to Charlie's chain see the same cross-reference in each block. However, chain owners may try to present different views to different users in different blocks by abusing the access-control mechanism. Thereby, the chain owner can equivocate between blocks.

Consider the following example, illustrated in Figure 12, in which the chain owner Owen shows Bob a fake cross-reference to Charlie's chain, while showing the correct cross-reference to Alice. To do so, he never lets Alice and Bob see claims about Charlie's chain in the same block. In block 1, he gives access to Alice, but not to Bob, while in blocks 2 and 3, he gives access to Bob, but not to Alice. Finally, in block 4, Owen again gives access to Alice but not Bob. If Owen has claims about Charlie's true chain in blocks 1 and 4—the ones that Alice can read—and false claims about Charlie's chain in blocks 2 and 3—the ones Bob can read—he is effectively launching an equivocation attack.

A trivial solution to prevent this attack would be to, upon suspicion, allow Alice and Bob to inquire about claims related to Charlie in the blocks where they do not have access. However, this can leak information about if and when the chain owner learned about Charlie's updates. To be able to withdraw the access while preventing the described attack in a privacy-preserving way, ClaimChain enables the chain owner to prove, in zero knowledge, that she did not equivocate in the blocks where the cross-references were not accessible by the reader.

Consider again our example in Figure 12. When Alice regains access to Charlie's references in block 4, she can use a detection mechanism to detect Owen's equivocation attempt. In other words, she can determine that in the intermediate blocks 2 and 3, where she did not have access to the cross-references about Charlie, Owen referenced a different chain than the one she sees. Bob would also detect the equivocation if he regains read access.

To enable detection, upon giving the access to Alice in block 4 again, Owen constructs a non-equivocation proof as follows.

1. Owen recomputes the VRF hashes $h_i = \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, l \parallel \mathsf{nonce}_i)$ for all intermediate blocks, and computes proofs of correctness $\pi_h^{(i)}$:

$$\pi_h^{(i)} = \mathsf{SPK}\{(\mathsf{sk}_{\mathsf{VRF}}) : \mathsf{pk}_{\mathsf{VRF}} = g^{\mathsf{sk}_{\mathsf{VRF}}} \wedge h_i = \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, l \parallel \mathsf{nonce}_i)\}()$$

   Alice can use the VRF hashes to locate the cross-reference to Charlie in the

procedure ProveConsistency($\mathsf{sk}_{\mathsf{VRF}}, l, \{O_i\}_1^n, \{C_i\}_1^t, \{(r_i, x_i)\}_1^t$)
  for $i = 1, \ldots, n$ **do**
    $h_i, \pi_h^{(i)} \leftarrow \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, l \parallel \text{nonce})$
    $\pi_{\text{ref}}^{(i)} \leftarrow \mathsf{SPK}\{(r_i, x_i) : \mathsf{com}_i = \mathsf{Commit}(r_i, x_i) \wedge x_i \in \{H_q(C_1), \ldots, H_q(C_t)\}\}()$
  return $\pi_{\text{consist}} = \{(h_i, \pi_h^{(i)}, \pi_{\text{ref}}^{(i)})\}_1^n$
procedure CheckConsistency($l, \{O_i\}_1^n, \{C_i\}_1^t, \pi_{\text{consist}}, \text{store}$)
  $\{(h_i, \pi_h^{(i)}, \pi_{\text{ref}}^{(i)})\}_1^n \leftarrow \pi_{\text{consist}}$
  for $i = 1, \ldots, n$ **do**
    Verify proof $\pi_h^{(i)}$ for h_i and $\mathsf{pk}_{\mathsf{VRF}}$
    Verify proof $\pi_{\text{ref}}^{(i)}$ w.r.t. $\{C_j\}_1^t$ and O_i using store.
  return $\top$ if all proofs verified, otherwise $\bot$

Figure 13: Proving and verifying that blocks O_i cross-reference the label l to the correct chain C_i.

intermediate blocks of Owen. The proofs $\pi_h^{(i)}$ confirm that she found the correct claims for Charlie's label l.

2. Owen proves in zero-knowledge that com_i commits to one of the intermediate blocks $C_1, \ldots, C_t$ on Charlie's chain:

$$\pi_{\text{ref}}^{(i)} = \mathsf{SPK}\{(r_i, x_i) : \mathsf{com}_i = \mathsf{Commit}(r_i, x_i) \wedge x_i \in \{H_q(C_1), \ldots, H_q(C_t)\}\}().$$

Owen compiles all the tuples $(h_i, \pi_h^{(i)}, \pi_{\text{ref}}^{(i)})$ and sends them to Alice. Alice uses these tuples to check that each of the intermediate blocks belong the same chain of Charlie that she saw before. If Owen indeed equivocated as in the example, Alice can detect this, since the proof verification would have failed. A detailed description of this procedure is given in Figure 13.

4.5 Evaluating the performance of ClaimChain

In this section we evaluate the performance of the new ClaimChain data structure. For the effectiveness of using ClaimChain for in-band public key distribution we refer to deliverable "D2.2: Federated Secure Identity Protocol".

4.5.1 Experimental setup

We implemented a prototype of ClaimChains in Python.[8] This implementation uses the petlib library [Dan] for elliptic curve cryptography operations, which internally relies on the OpenSSL C library. For the implementation of hash chains and unique-resolution Merkle trees we use the hippiehug[9] library, which is written in pure Python. Our implementation uses AES128 in GCM mode for symmetric encryption; ECDSA, ECDH, and other elliptic curve operations with a SECG curve over a 256 bit prime field ("secp256k1"); and SHA256 as the base hash function.

[8] https://github.com/claimchain/claimchain-core
[9] https://github.com/gdanezis/rousseau-chain

Table 3: ClaimChain basic operations timing

	mean (ms)	std. (ms)
Label capability lookup key computation	0.30	0.01
Label capability decoding	0.33	0.01
Label capability encoding	0.33	0.02
Claim encoding [π computation]	2.44 [2.38]	0.05 [0.05]
Claim decoding [π verification]	3.03 [2.96]	0.05 [0.05]

All the lookup keys on the claim map are truncated to 8 bytes, which makes collisions unlikely for up to 2^{32} entries in the map. The size of the per-block nonce is set to 16 bytes, and it is generated using the standard Linux `urandom` device.

Our experiments are also publicly available and reproducible.[10] We extensively use Jupyter notebooks [Klu+16] and GNU parallel [Tan11]. We run the experiments on an Intel Core i7-7700 CPU @ 3.60GHz machine using CPython 3.5.2.

4.5.2 ClaimChain operations performance

We now evaluate the performance of ClaimChains in terms of computation time and storage.

Timing. We first measure the computation time for encoding and decoding claims and capabilities as described in Section 4.3.3. We encode and decode 1000 claims and corresponding capabilities for random readers (i.e., encoded for a random DH public key). Each claim has a 32-byte random label and 512-byte random content. This reflects a realistic e-mail setting: 32-byte labels can accommodate e-mail addresses or their hash; and 512 bytes approximates the approx. 500-bytes block size in our experiments below.

Table 3 reports our measurements. The time for encoding, decoding, and computing lookup keys for capabilities is under 0.33 ms. The time to encode and decode claims is around 3 ms, consisting mostly of the proof computation and verification time.

The most computationally expensive operation that ClaimChain owners perform is constructing the block map when a new block is created. The map is constructed using the BuildTree procedure (see Algorithm 1 in Appendix A.1). We measure the time to create a block map of n claims with one capability each, i.e., readable by only one reader. We range n from 100 to 5,000. For each case we construct a unique-resolution key-value Merkle tree with the encoded entries. Figure 14 (left) shows the average time required to build the tree across 20 experiments. Even for 5,000 claim-capability pairs the operation takes under 0.3 seconds. In reality, we expect users to have much fewer entries per block (in our simulation using the Enron dataset this number rarely exceeds 1,000).

Recall that along with the block, users send paths that prove the inclusion of relevant claims and capabilities in the block map tree. These are computed using the GetPath procedure (see Algorithm 2 in Appendix A.1). We measure the time to compute and verify a proof for a single entry, as well as the proof size in terms of number of tree nodes and bytes. We use the same setting as in the previous experiment. Unsurprisingly, the computation and verification time, and the proof

[10] https://github.com/claimchain/claimchain-simulations

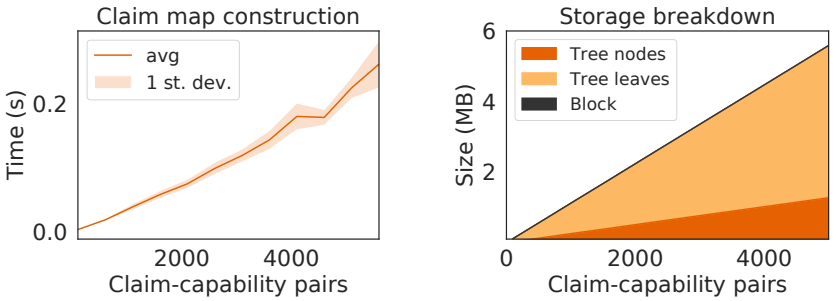


Figure 14: Total storage size and claim map construction time

size scale logarithmically with the number of items in the map. For 5,000 items, computation and verification take on average about 150 milliseconds, and the proof consists of on average 20 tree nodes and takes about 1.5 KB.

Storage. We measure the size of a ClaimChain block, a block map tree, and values stored in the leaves of the tree (encrypted claims and capabilities). The size of the block map depends on the number of entries in the map and the size of claims. Figure 14 (right) shows the size breakdown depending on the number of items in the map. Note that the block itself only includes the root of the tree. Thus, the block size is constant (about 500 bytes), and can only grow if security parameters change (size of cryptographic public keys, or hash length increases), or additional data about the owner is added.

Inter-block equivocation detection. The cost of proving consistency is dominated by the proof $\pi_{\text{ref}}^{(i)}$. Using a straightforward instantiation with 'or' proofs, the prover and verifier must compute approximately $5t$ exponentiations to construct and verify $\pi_{\text{ref}}^{(i)}$, where t is the number of possible cross-referenced blocks. Therefore, a full consistency proof requires approximately $5nt$ exponentiations, where n is the number of intermediate blocks on the owner's chain.

4.6 Concluding remarks

In-band key distribution, as proposed by Autocrypt, is a promising direction towards achieving e-mail encryption without the collaboration from service providers. However, it suffers from security and privacy problems. To address these issues we introduced ClaimChains, a construction that can be sent in-band with e-mails to provide high-integrity evidence of key-identity bindings. Its cryptographic access control enables users to selectively reveal their contacts, preserving their privacy, while preventing equivocation attacks in which different users are shown different bindings.

We demonstrate that key propagation, and thus the ability to encrypt messages, is not affected much when using the privacy features of ClaimChains. However, users do obtain less evidence about other users' bindings, increasing the chances that wrong keys go unnoticed. On the negative side, our study shows that the coverage achieved by in-band key distribution is partial at best. In our realistic simulations we could achieve a maximum of 66% of e-mail encrypted, even within a well-connected social network.

However, we note that the design of ClaimChains is not tied to decentralized stor-

age and distribution. Their strong security and privacy properties permit to host the content-addressable storage in semi-trusted providers without relying on them to return correct values. Such deployment of ClaimChains would greatly improve availability of ClaimChain data. But, to obtain perfect privacy, such scheme requires integration with privacy-preserving storage access [Cho+95; TDG16] to avoid leakage stemming from access patterns.

Finally, ClaimChain or its component data structures can have applications to use cases beyond key distribution. The claim map data structure, for example, can be applied in similar settings when a verifiable dictionary with cryptographic access controls for its lookup keys is needed.

5 Tandem

The security and privacy properties of cryptographic schemes, including Claimchains, rely on the security of the underlying cryptographic keys. As a result of decentralization, these keys move to the user's devices, which are difficult to secure.

In this section we present Tandem, a system that allows users to increase the security of their keys by sharing a part of their cryptographic keys with a central server *without* having to give up any of their privacy to do so.

5.1 Introduction

The security of cryptographic schemes hinges on the security of the underlying keys. However, secure solutions to store and process keys on users' devices are hard to deploy in practice. Software-based approaches are extremely difficult to secure [Her15; Vee+16; Kim16; Lip+18], and secure hardware [EKA14; Mar+13; SZ05] might not be available on users' devices, not accessible to developers [McG+15; And17], or harmful to usability [DDC18].

As an alternative, users could use a secure central server to store their keys and perform cryptographic operations on their behalf, and to block their keys if their devices are compromised. The problem is that centralization introduces security and privacy concerns that are not an issue when keys are stored on the user's device. First, users must trust the central server to not impersonate them. Second, the central server is in a privileged position to learn private information about users from their interactions with other services. Brandão et al. illustrate these problems in the context of nation-scale brokered-identification systems [Bra+15]. They show how a central hub that acts as the broker between users, identity providers, and service providers can impersonate users, link users' transactions across different service providers, and also learn private identifiable information about users.

A natural solution to the impersonation problem is to involve the user in the storage and/or usage of the keys by using threshold cryptography. This approach additionally strengthens authentication security as the user needs a second factor: a key share. However, threshold cryptography does not address the privacy concerns associated with centralization. The central server learns the users' key-usage patterns and, as the time of access and use of the key are almost the same, it can use this information to deanonymize users' anonymous transactions, e.g., correlating interactions to public activities such as updates to a blockchain ledger [Jaw+18; Gol+18].

In this work, we present TANDEM, a set of protocols that augment threshold-cryptographic schemes to enable secure and privacy-preserving usage of key shares managed by a central server. To use a key, a user sends a *one-time-use key-share token* to the central server using an anonymous communication channel. This token contains a randomized version of the central server's key share for this user. The server uses this key share to run the threshold-cryptographic protocol *without* learning the user's identity.

The construction of key-share tokens permits to decouple the stages of obtaining and using the tokens, eliminating the possibility of time-correlation attacks. Furthermore, the one-time property enables two additional functionalities: the blocking of keys in case the user's key share is compromised, and the rate limiting of key usage to restrict how often an attacker can use an unblocked key. TANDEM provides these functionalities *without* the need to identify token owners.

TANDEM can be used to secure the keys of any cryptographic scheme (e.g., encryption, signature, or payments) for which a linearly randomizable threshold-cryptographic version of the scheme exists. As long as the threshold version is private, i.e., the scheme does not require information that identifies the user besides the key, TANDEM ensures that not even a malicious central server can learn with which user it is interacting. For example, Tandem can be applied to threshold variants of Schnorr [Gen+07] and RSA signatures [Sho00], ElGamal-based [ElG84; SG02] and RSA decryption [Sho00], as well as threshold-cryptographic versions of electronic cash schemes [CHL05; Mie+13] and attribute-based credential schemes [ASM06; Bra00; CH02; CL02]. We note, however, that Tandem cannot be applied to existing threshold DSA schemes because they are multiplicative [MR04] or require identifying auxiliary information [GGN16].

To demonstrate the potential of TANDEM we use it to secure keys in a threshold version of BBS+ attribute-based credentials (ABCs). ABCs [ASM06; Bra00; CH02; CL02] protect users' anonymity during authentication on sensitive online services, e.g., online health services[11]. Thus, enhancing them with a naïve centralized approach in which the central server could learn which sensitive services users access would defeat the very purpose of ABCs. Using TANDEM to secure her keys, the user can show her credential to the online service *without* the TANDEM server learning who is using the key, preserving the user's privacy even if the TANDEM server and the service provider collude. Moreover, the user never has the complete key in her device.

The anonymity provided by ABCs opens the door to malicious users abusing service providers. We also provide a simple modification to the threshold ABC schemes that enables service providers to confirm that TANDEM is used. Then, as long as all users use TANDEM, TANDEM can replace the complex ad-hoc cryptographic techniques to block users [ATK11; Tsa+10] or limit key-usage [Cam+06].

We validate the practicality of the TANDEM protocols on a prototype C implementation. Using a key with TANDEM induces a 50 – 100 ms overhead on the TANDEM server with respect to traditional threshold-cryptographic solutions, and only 5 ms overhead on the user. The cost for the server is manageable. On the user side, the overhead is negligible with respect to the delay imposed by the use of anonymous communications necessary for typical uses of TANDEM such as anonymous web-based authentication.

[11] such as `https://medical.mit.edu/services/mental-health-counseling` and `https://www.nhs.uk/Conditions/online-mental-health-services/Pages/introduction.aspx`

In summary, we make the following contributions:

✓ We introduce TANDEM; it enables the use of threshold-cryptographic protocols with a central server to secure cryptographic keys without this server learning what keys are used by whom. Additionally, TANDEM enables blocking and rate limiting of key usage.

✓ We provide a threshold version of an attribute-based credential system, and show how TANDEM can be used to augment its security. We show how the underlying constructions in TANDEM permit rate limiting and revocation of credentials *without* relying on complex purpose-built cryptographic techniques.

✓ We prove the security and privacy of TANDEM, and we use a prototype implementation to validate its practicality. All operations in TANDEM take less than one second, imposing a reasonable overhead on both server and users.

5.2 Related Work

Exisiting solutions to protect cryptographic keys fall into two coarse categories, either *single-party* or *decentralized*. The former typically rely on secure hardware [Mit05; SZ05; Mar+13; EKA14] that securely stores and processes cryptographic keys within the secure environment. However, secure hardware is expensive, is not always available (e.g., in laptops) or not accessible by application developers, and is often not flexible enough to run advanced protocols.

Threshold cryptography aims to strengthen general cryptographic protocols by distributing the user's secret key among several parties. This approach was first proposed by Desmedt [Des87] and Boyd [Boy89]. Several threshold encryption and signature schemes have been proposed since then [DF91; Rab98; Gen+00; Gen+07; Sho00; ADN06; PNP08; Haz+12]. More recently, Atwater et al. [AH16] built a library to execute such protocols in users' personal devices. Other works have tackled more complicated protocols. For instance, Brands shows how to distribute the user's secret key in attribute-based credentials [Bra00], and Keller et al. [KMR12] show how to make threshold-cryptographic versions of zero-knowledge proofs.

Many works propose systems in which the user's secret key is shared between a user's device and a central server to protect the key and also to enable instant blocking of the key by the user [MR01; Cam+16; Bon+01; BDT04; LQ03; Bul+17]. However, none of these schemes provide privacy for the user towards the central server. In all of these schemes users authenticate themselves to the server, making them susceptible to time-correlation attacks [Jaw+18]. Camenisch et al. [Cam+16] attempt to ensure privacy to some extent in signature schemes by blinding the message being signed during the threshold protocol with the server. Yet, the server learns when and how often the user uses her signing key. Hence, users are still vulnerable to timing attacks. The scheme by Brands [Bra00] protects against these attacks as long as the key-share holder is a smartcard, which cannot store a timed log of operations. However, if the smartcard is replaced by an online server that holds the key share, this server learns the key-usage patterns of users. Then, the cryptographic measures proposed by Brands alone cannot prevent time-correlation attacks.

TANDEM is designed to complement these threshold-cryptographic solutions to make them privacy friendly. We compare the privacy properties obtained when using TANDEM with those in previous proposals in Table 4. We consider three privacy aspects: anonymity of the user when running the threshold protocol (i.e., need

Table 4: Comparison of generic and special-purpose TCPs with TCPs augmented with Tandem.

	Generic e.g., [Gen+07; SG02]	Purpose [Cam+16; Bra00]	with TANDEM
Anonymous key usage	×	×	✓
Hide protocol data	×	✓	×*
Hide key-usage patterns	×	×	✓

* This property is irrelevant for TANDEM; see text.

to authenticate); hide the data used (e.g., signed message) in the protocol from the server; and hide the usage pattern to avoid timing attacks. Generic schemes focus on achieving security of the secret key and thus provide no privacy. Special-purpose designs, have so far only focused on the protection of data involved in the protocol. TANDEM does not need to protect this data: the other two properties decouple the data from the user's identity.

Finally, similarly to TANDEM, password-hardening services [Eve+15; CLN15] use decentralization to increase security against brute-force attacks. These schemes introduce a hardening server that rate limits, or even blocks, requests from the main authentication server. However, the cryptographic techniques behind these solutions cannot be directly applied to the problem tackled by TANDEM. First, they are designed for a particular task: securely verifying passwords, and adapting them to run other protocols is non-trivial. Second, in the password scenario the hardening server is only accessed by the authentication server. Therefore, there are no privacy concerns, and these techniques do not provide any privacy protection.

5.3 Problem Statement

We consider a scenario in which *users* are required to perform cryptographic operations to interact with a *service provider* (SP). Users use insecure devices, such as smartphones, tablets, or laptops, without secure hardware, to run the cryptographic protocols. To keep their keys safe, they use a centralized *TANDEM Server* (TS) to run threshold-cryptographic protocols (TCPs) in a distributed way. Users wish to keep their key-use pattern and their use of other services private with respect to the TS. We call an execution of the protocol between the user and the TS a *transaction*.

For simplicity, we assume that there is only one TANDEM server. However, we note secret-sharing the key with multiple TANDEM servers would increase security. In this case TANDEM would ensure that keys can be blocked and rate limited as long as at least one of the TANDEM servers is honest. Privacy is not affected by the number of servers.

5.3.1 TANDEM Properties and Threat model

PROPERTY 1 (KEY SECURITY). TANDEM *protects the use of the user's key.* No entity other than the user is able to use the user's key. Even if the user's device is compromised, the user can maintain this property by *blocking the key* at the TANDEM server. Thereafter, the attacker cannot further use her key. We formalize this property in Game 1 in Section 5.6.

Any solution that recomputes the full user's key on the user's device, e.g., by deriving it from a user-entered passphrase, does not satisfy this key-security property. In such a solution, an attacker who compromises the user's device can observe the full key when it is used. Thereafter the attacker can use the key indefinitely, making blocking impossible.

Property 2 (Key rate-limiting). Tandem *limits the rate of usage of keys.* Users can limit the number of times her key is used in a given interval of time. We call this interval an *epoch*. We formalize this property in Game 2 in Section 5.6.

The security and rate-limiting properties are related to the revocation and n-times-use concepts of attribute-based credentials [Cam+06], respectively. Yet, they are not the same. Revocation and n-times-use credentials trust the service providers to block credentials respectively to block a credential after n uses. Using Tandem on the other hand, users need to trust only the TS, which *they* choose, to block and rate-limit keys. Tandem can ensure this property for a large class of protocols, even if a system does not rely on credentials.

Property 3 (Key-use privacy). Tandem *protects the privacy of key use in transactions.* The Tandem server (TS) cannot distinguish between two users performing transactions. Even if the TS colludes with the service provider (SP) it cannot distinguish users (unless the SP could distinguish the users, in which case collusion leads to a trivial and unavoidable privacy breach). We formalize this property in Game 4 in Section 5.6.

We assume that the Tandem server is honest with respect to security. That is, it follows the protocols so as to protect the security of users' keys (Property 1) and to ensure that keys are only used the allowed number of times (Property 2). Moreover, we trust the Tandem server to be available, i.e., Tandem does not protect from denial of service. However, the Tandem server may be malicious with respect to privacy: It is interested in breaching the privacy of the users by trying to learn which keys and services they use (Property 3).

Why naïve solutions do not work. Consider an approach in which a user naïvely secret-shares her key with the Tandem server. When she needs to run a threshold-cryptographic protocol, the user authenticates to the TS, the TS recognizes the user, retrieves its share of the user's key, and executes the TCP together with the user. This scheme offers key security (Property 1): The TS alone cannot use the user's key, and if an attacker compromises the user's device, the user can authenticate to the TS and request it to block her key. This scheme also provides key rate limitation (Property 2): the TS can observe when a user accesses her key. Hence, it can easily enforce a limit on the number of times the key is used. However, since the user is identified while using the key for a TCP, the scheme does not achieve key-use privacy (Property 3).

The lack of key-use privacy has further implications when the interactions between the user and the SP are anonymous (e.g., showing an anonymous credential). The SP can collude with the TS to learn the user's identity, exploiting the fact that there is a strong correlation between the time when the authenticated user interacts with the TS, and when the anonymous user interacts with the SP. Thus, for every anonymous transaction with the SP, the anonymity set of the user is reduced to the authenticated users interacting with the TS around the transaction time. This attack has been used in the early days of Tor to identify users and hidden services [Abb+07; ØS06]. The attack relies solely on time correlation between accesses. Therefore, the attack cannot be prevented by making the messages seen by the TS and the SP during the TCP cryptographically unlinkable [Bra00].

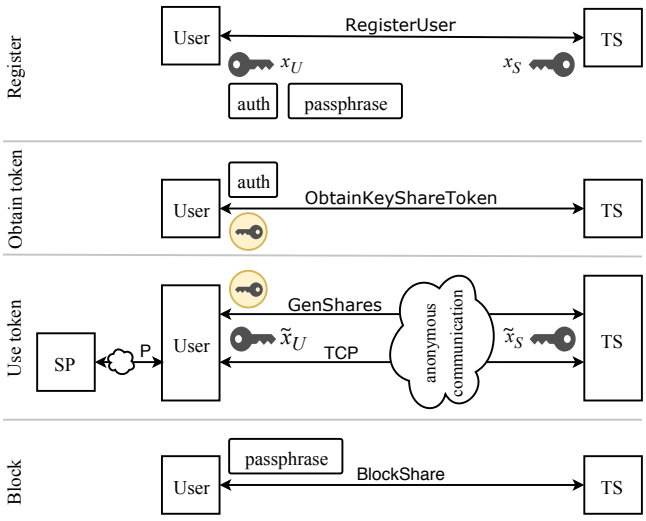


Figure 15: TANDEM process: after registration, a user can authenticate herself and obtain key-share tokens, which can later be used anonymously to execute a TCP (the user and SP run protocol P). The user can block her keys at any time. Inputs are shown above the arrows, outputs below.

There are two straightforward approaches to prevent time-correlation attacks: introducing delays and introducing dummy requests. These solutions are, however, difficult to use in practice. To significantly increase the anonymity set for users, operations may need to be delayed for a long time. This rules out applications that require short delays, such as showing an anonymous credential or performing a payment. Dummy traffic not only imposes an overhead on users and the TS, but it is widely known that generating dummy actions that are indistinguishable from real activity is very difficult [BTD12; CG09]—especially because it is unrealistic that users would be always online so that their devices could produce such requests.

5.3.2 TANDEM at a Glance

We now provide a high-level overview of how users can use the TANDEM server to perform threshold-cryptographic protocols in a privacy-preserving way, see Fig. 15. We assume that users can use an anonymous communication channel [Pio+17; DMS04b] to communicate with the TANDEM server (TS) and service providers (SPs) to protect their privacy at the network layer.

Registration. First, users register with the TS using the RegisterUser protocol. During registration, the user and TS jointly compute long-term shares x_U and x_S of a long-term key x appropriate for the threshold-cryptographic protocols they seek to run later. The user obtains credentials to authenticate when obtaining a token (e.g., a password) and also a means to block her keys (e.g., a passphrase); she stores the latter outside her device.

Obtain Token. Key-share tokens enable the user to anonymously use her key later (see below). To obtain a token, the user runs the ObtainKeyShareToken protocol with the TS. First, the user authenticates herself to the TS. Then, the user and the TS construct a one-time-use key-share token, containing a randomized version of the TS's key share x_S. At this stage, the TS can limit the number of tokens it provides the user, thus limiting how often the user can use her key.

Key-share tokens may seem similar to passwords: both unlock functionality. However, unlike passwords, key-share tokens can be verified and used *without* knowing the user's identity. Moreover, key-share tokens contain a randomized key share $\tilde{x}_S$ essential for the TCP. Hence, TANDEM cannot be replaced by a password-hardening service [Eve+15; CLN15]. The randomized key shares contained in the tokens also distinguish the tokens from traditional eCash tokens [CFN; CHL05; Mie+13].

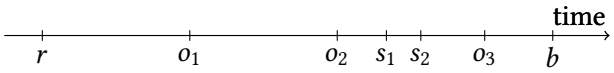
Using Keys. After obtaining tokens, a user can run threshold-cryptographic protocols with the TS. First, the user and the TS use the token to derive fresh shares $\tilde{x}_U$ and $\tilde{x}_S$ by running the GenShares protocol. The new share $\tilde{x}_S$ cannot be linked to the long-term share x_S, thus it does not reveal the user's identity to the TS. The user and the TS use the fresh shares $\tilde{x}_U$ and $\tilde{x}_S$ as input to the threshold-cryptographic protocol TCP, allowing the user to use her key in the cryptographic protocol P with the service provider.

The TS never communicates directly with service providers, but only via the user. Therefore, the use of TANDEM can remain invisible to the service provider, i.e., users can use TANDEM without the SP's knowledge or consent.

Blocking keys. Whenever a user wants to block her key, she requests the TS to block her key by using the BlockShare protocol with her blocking means (e.g., the passphrase). Thereafter, unused tokens become invalid, and no new tokens can be obtained (not even by an adversary that knows the authentication credential used to obtain tokens). Hence, the user's key cannot be used anymore.

Preventing Time Correlation. When obtaining tokens, the user is authenticated. Hence, to preserve privacy, the actions of obtaining and using tokens *must* be uncorrelated, i.e., tokens should not be obtained right before usage. To avoid correlation, the user can configure her device to obtain tokens at random times or at regular times (e.g., every night) such that tokens are always available. The user can authenticate herself to the device at those times, or automate the process by storing her authentication credential on the device. Note that the user's key can still be blocked at the TS if an attacker learns this authentication credential.

Here we show an example time line of registration (r), obtaining tokens (o_i), using tokens (s_i), and blocking the key (b) events:



This example illustrates that obtain and use events do not necessarily follow each other, but can be interleaved. As a result, the timing of these events needs *not* to be correlated. The token o_3, unused before the key is blocked by b, cannot be used after time b.

In Section 5.5.2 we explain why private information retrieval is not a suitable alternative to decouple obtaining and using of key shares, and how TANDEM outperforms generic alternatives based on secure multi-party computation.

5.4 Cryptographic preliminaries

Let ℓ be a security parameter. Throughout this paper, $\mathbb{G}$ is a cyclic group of prime order p (of 2ℓ bits) generated by g. We write $\mathbb{Z}_p$ for the integers modulo p. We use a cryptographic hash function $H : \{0,1\}^* \to \mathbb{Z}_p$ that maps strings to integers modulo

p. We write $a \in_R A$ to denote that a is chosen uniformly at random from the set A. Furthermore, we write $[n]$ to denote the set $\{0, \ldots, n-1\}$.

For reference, Table 5 in Section 5.5 summarizes the notation used by TANDEM's building blocks, and Table 6 in Section 5.5 explains frequently-used symbols in TANDEM.

5.4.1 Cryptographic Building Blocks

TANDEM relies on a couple of cryptographic building blocks. First, we use an additive homomorphic encryption scheme given by the algorithms $\mathbf{E}^+_{pk}, \mathbf{D}^+_{sk}$ with plaintext space $\mathbb{Z}_N$ (i.e., integers modulo N) and space of randomizers $\mathcal{R}$. We write $c = \mathbf{E}^+_{pk}(m; \kappa)$ to denote the homomorphic encryption of the message $m \in \mathbb{Z}_N$ using randomness $\kappa \in \mathcal{R}$. The scheme is additively homomorphic, so

$$\mathbf{E}^+_{pk}(m_1; \kappa_1) \cdot \mathbf{E}^+_{pk}(m_2; \kappa_2) = \mathbf{E}^+_{pk}(m_1 + m_2 \pmod N; \kappa_1\kappa_2).$$

Our proof of concept uses Joye and Libert's encryption scheme [JL13], but Paillier's scheme [Pai99] would also work.

Second, TANDEM uses a CPA secure encryption scheme $\mathbf{E}_{pk_{id}}, \mathbf{D}_{sk_{id}}$ with plaintext space $\mathbb{G}$ such as ElGamal [ElG85], that allows simple verifiable encryption.

Third, TANDEM uses two computationally hiding and binding commitment schemes. First, by $\mathsf{Commit}(m, r)$ we denote a commitment function that takes a message $m \in \mathbb{Z}_p$ and a randomizer $r \in \mathbb{Z}_p$. Analogously, we define $\mathsf{Commit}((m_1, \ldots, m_k), r)$ to commit to a tuple of messages. We instantiate this scheme using Pedersen's commitments [Ped91], because it enables users to obtain a blind signature on the tuple $(m_1, \ldots, m_k)$. However, any other commitment scheme with these properties suffices as well.

Second, we denote by $\Delta = \mathsf{ExtCommit}(m, r)$ with $m \in \{0,1\}^*, r \in \{0,1\}^{2\ell}$ an extractable commitment scheme. That is, in our security reductions, we can extract the input m used to create a commitment Δ. For example, the instantiation $\mathsf{ExtCommit}(m, r) = H(m \| r)$ is extractable in the random oracle model.

5.4.2 Threshold-Cryptographic Protocols

In this paper, we focus on cryptographic protocols run between a user and a service provider, e.g., showing a credential to an SP or spending an electronic coin. The threshold-cryptographic version of such a protocol splits the user's key x and the user's side of the original protocol in two parts, run by different parties. Each party operates on a secret-share of the user's key. Security of the threshold-cryptographic protocol (TCP) ensures that a large enough subset of shares (two in the case of two parties) are required to complete the protocol.

We consider TCPs where the user's side of the protocol is distributed between the user and the TS. After registration, the user and the TS hold the key shares x_U and x_S of x, respectively. After running GenShares, the user and the TS hold the fresh key shares $\tilde{x}_U$ and $\tilde{x}_S$. They then run the TCP protocol, which we denote as:

$$\mathsf{P}(\mathsf{in}_{SP}) \leftrightarrow \mathsf{TCP.U}(\tilde{x}_U, \mathsf{in}_U) \leftrightarrow \mathsf{TCP.TS}(\tilde{x}_S), \tag{1}$$

where the SP, the user and the TS respectively run the interactive programs P, TCP.U and TCP.TS. The user mediates all interactions between the service provider and the

TS. The user and the SP take extra inputs needed for the execution of the target cryptographic protocol denoted as in_U and in_{SP}. For simplicity, we denote the complete protocol from (1) by $\mathsf{TCP}(\tilde{x}_U, \tilde{x}_S, \mathsf{in}_U, \mathsf{in}_{SP})$.

Tandem can only enhance the privacy (Property 3) of certain TCPs. We formalize the condition that these TCPs should satisfy. To avoid that the TS can recognize the user based on the shares input to the TCP, we randomize the long-term secret shares. Thus, we require that TCPs enhanced with Tandem still function with randomized key shares. In addition, our privacy-friendly GenShares protocol requires this randomization to be linear.

For simplicity, we assume that the user's secret $x \in \mathbb{Z}_p$ for some field $\mathbb{Z}_p$ of prime order p (e.g., corresponding to the group $\mathbb{G}$ we defined above). We note, however, that our constructions can be modified to settings with unknown order arising from RSA assumptions. Formally, we require the TCP to be linearly randomizable:
Definition 1. Let $x_U, x_S \in \mathbb{Z}_p$ be secret shares of the user's secret x. Then, we say that the TCP is *linearly randomizable* if for all δ we have that (1) if $\mathsf{TCP}(x_U, x_S, \mathsf{in}_U, \mathsf{in}_{SP})$ completes successfully, then so does $\mathsf{TCP}(x_U - \delta, x_S + \delta, \mathsf{in}_U, \mathsf{in}_{SP})$, and (2) $x_S + \delta$ is independent from x_S.

The first condition implies that the original secret sharing (x_U, x_S) and the randomized secret sharing $(x_U - \delta, x_S + \delta)$ must share the same secret, whereas the second implies that the TS cannot recognize the user from the randomized secret share alone.

Security and privacy properties of TCPs. To ensure that a TCP with Tandem satisfies the security properties (Property 1 and Property 2) we require that the TCP itself is secure. That is, if the TS no longer uses its share x_S to run its part of the TCP, then no malicious user can successfully complete the TCP with the SP. We formalize this notion in Game 3 in Section 5.6.

To ensure that a TCP with Tandem satisfies the privacy property (Property 3) we require that the TCP itself offers privacy with respect to the TS respectively the TS and the SP. That is, if the TS runs its part of the TCP using a randomized key-share as input, then the TS respectively the TS and the SP cannot recognize the user. We formalize this notion in Game 5 in Section 5.6.

5.5 Tandem

In this section, we present a construction that enables anonymous users to use their keys with the TS *without* the TS learning which key is being accessed. It uses homomorphic encryption to decouple the action of accessing the user's long-term key-share x_S at the Tandem server from its subsequent use in the threshold-cryptographic protocols. Thus, it prevents time-correlation attacks.

Initially, the TS generates a private-public key pair (sk, pk) for an additively homomorphic encryption scheme (see Section 5.4). The TS publishes the public key pk. Upon registration with the TS, a user receives $\overline{x_S} = \mathbf{E}^+_{pk}(x_S)$—a homomorphic encryption of the TS's key-share x_S. Because the ciphertext $\overline{x_S}$ is encrypted against the TS' key, the user does not learn anything about the TS' share x_S.

When the user wants to *use* her key, she produces a randomized version of the TS' key-share x_S. To produce this randomization, she picks a large δ and computes $c = \overline{x_S} \cdot \mathbf{E}^+_{pk}(\delta) = \mathbf{E}^+_{pk}(x_S + \delta)$. On her side, she randomizes her key as $\tilde{x}_U = x_U - \delta$

(mod p). Then, she sends c to the TS via an anonymous channel. The TS decrypts c to recover its key for the threshold cryptographic protocol, $\tilde{x}_S = x_S + \delta \pmod{p}$. It is easy to see that a linear TCP with randomized shares completes successfully, because $\tilde{x}_S + \tilde{x}_U = x_U + x_S \pmod{p}$. Because c is randomized, the TS can no longer recognize its share x_S, effectively decoupling this action from the key-share generation.

In this approach, however, the TS cannot block or rate-limit keys. We present below a construction for one-time-use key-share tokens containing signed and randomized ciphertexts like c that enables blocking and rate-limiting while preserving users' privacy.

5.5.1 One-time-use Key-share Tokens

To construct a token the user picks a large δ and homomorphically computes $c = \overline{x_S} \cdot \mathbf{E}^+_{pk}(\delta)$, a randomized encryption of the TS' key share. Then, she sends a commitment to c to the TS, together with a proof that the committed c was constructed by additively randomizing $\overline{x_S}$. This proof is needed to enable secure blocking as we explain below. The user engages with the TS to obtain a blind signature σ on c. The signature σ is only known to the user at this stage. The user stores the token $\tau = (\sigma, c)$ and the randomizer δ.

To run a threshold-cryptographic protocol the user anonymously contacts the TS and sends her key-share token $\tau = (\sigma, c)$. The TS checks the signature and makes sure the token was not used before. Then, the TS recovers the randomized key-share $\tilde{x}_S = \mathbf{D}^+_{sk}(c) \pmod{p} = x_S + \delta \pmod{p}$ and uses it as the key for the threshold cryptographic protocol. The user, on the other hand, uses $\tilde{x}_U = x_U - \delta \pmod{p}$ as the key. As in the previous case, because c is fully randomized, the TS cannot leverage it to identify users. Moreover, as σ is a blind signature on c the TS cannot use σ or c to link the token creation to the token use.

When a user asks the TS to block her key, the TS no longer creates key-share tokens for this user (we explain how the TS blocks unspent tokens below). This prevents attackers from further running threshold-cryptographic protocols, even if they corrupt the user's device. For the blocking of keys to be effective, attackers must not be able to construct key-share tokens for a blocked user. Here is where the proof becomes handy that c is constructed as $\overline{x_S} \cdot \mathbf{E}^+_{pk}(\delta)$, where $\overline{x_S}$ belongs to the current user. Suppose that we omit the proof. Then, an attacker controlling an unblocked user can create tokens for a corrupted blocked user. The attacker uses the unblocked user's account to make the TS blindly sign encrypted key shares for the blocked user. The attacker can use the resulting token to use the blocked user's key, defeating the purpose of Tandem. Verifying which user's key share is embedded into the ciphertext blindly signed by the TS prevents the attack.

Finally, since tokens are one-use only, to restrict the number of times a user can use her key (rate-limit), the TS just signs a limited number of key-share tokens per-epoch per-user.

Registering Users. When a user first registers at the TS, the TS computes a key-share x_S for that user, and sends her an encrypted version $\overline{x_S} = \mathbf{E}^+_{pk}(x_S)$. To ensure that the TS cannot hide an identifier in higher-order bits of x_S that are not randomized by the user in the remainder of the protocol the TS proves that the plaintext x_S is in the correct range.

Table 5: Notation and cryptographic building blocks used by TANDEM.

Symbol	Interpretation
$[n]$	The set $\{0, \ldots, n-1\}$
ℓ	The security parameter
$\mathbb{G}, g, p$	Cyclic group $\mathbb{G} = \langle g \rangle$ of order p
Additively homomorphic encryption scheme	
$\mathbf{E}^+_{pk}(m; r)$	Encrypt message $m \in \mathbb{Z}_N$ with randomizer $r \in \mathcal{R}$
$\mathbf{D}^+_{sk}(c)$	Decrypt ciphertext c
N	Size of additive plaintext domain
$\mathcal{R}$	Space of randomizers
CPA secure verifiable encryption scheme	
$\mathbf{E}_{pk_{id}}(m)$	Encrypt message $m \in \mathbb{G}$
$\mathbf{D}_{sk_{id}}(c)$	Decrypt ciphertext c
Commitment schemes and hash function	
$\mathsf{Commit}(m, r)$	Commit to $m \in \mathbb{Z}_p$ (or a tuple of messages) with randomizer $r \in \mathbb{Z}_p$
$\mathsf{ExtCommit}(m, r)$	Commit to $m \in \{0,1\}^*$ with randomizer $r \in \{0,1\}^{2\ell}$
$H(s)$	Hash function from $s \in \{0,1\}^*$ to $\mathbb{Z}_p$

PROTOCOL 1. The RegisterUser protocol is run between a user and the TS, and proceeds as follows.

1. The user U and the TS generate secret shares $x_U \in_R \mathbb{Z}_p$ and $x_S \in_R \mathbb{Z}_p$, respectively. The user also generates a public-private key-pair (pk_{id}, sk_{id}) for encrypting token identifiers and sends pk_{id} to the TS. The user needs the secret key sk_{id} to block unspent tokens if needed. We assume that the user stores sk_{id} externally so that it is available even after she loses her device. We propose that the user's device generates sk_{id} based on a high-entropy passphrase (such as a Diceware passphrase[12]), so that users can write down this string as a stand-in for sk_{id}.

2. The TS picks $\kappa \in_R \mathcal{R}$, computes $\overline{x_S} = \mathbf{E}^+_{pk}(x_S; \kappa)$ and sends $\overline{x_S}$ to the user. Moreover, the TS sends a range proof to the user that $\overline{x_S}$ is constructed correctly, i.e., that
$$\mathbf{D}^+_{sk}(\overline{x_S}) \in [0, p). \tag{2}$$
See Appendix B.2 for how to instantiate this proof.

3. The TS records $(x_S, \overline{x_S}, pk_{id})$ for this user, and marks this user as active. The user stores $(x_U, \overline{x_S}, pk_{id})$ on her device, and stores sk_{id} externally.

Obtain a Key-share Token. First, the user needs to randomize the ciphertext $\overline{x_S}$. However, it seems difficult to prove directly, for example in zero-knowledge, that the randomized ciphertext produced by the user is of the correct form. Therefore, we use a standard cut-and-choose approach [BCC88; CFN] to allow the TS to check that the encrypted key share it is blindly signing is correct with overwhelming probability. The user constructs $2k$ randomized ciphertexts $c_i = \overline{x_S} \cdot \mathbf{E}^+_{pk}(\delta_i; \kappa_i)$, and sends commitments C_i to them to the TS. The TS then asks the user to open a subset $\mathcal{D}$ of cardinality k, so that the TS can verify that these k ciphertexts were correctly formed. Having checked all opened ciphertexts, the TS blindly signs the remaining

[12] http://world.std.com/~reinhold/diceware.html

Table 6: Notation in TANDEM protocols

Symbol	Interpretation
$\mathcal{D}$	Disclose subset in cut-and-choose construction
δ, δ_i	Randomizers of key shares
id	Token identifier
k	Token security parameter
ℓ_δ	Length of randomizers δ_i in bits
x	Long-term secret key for a user
pk, sk	Public-private key-pair of TS
pk_{id}, sk_{id}	Public-private key-pair of the user U
p	Order of the group $\mathbb{G}$
x_U	Long-term key share held by the user
x_S	Long-term key share held by the TS
$\overline{x_S}$	Homomorphic encryption of x_S
$\tilde{x}_U$	User's key share output by GenShares
$\tilde{x}_S$	TS' key share output by GenShares
ϵ	The current epoch
σ	Blind signature of the TS

k ciphertexts. By nature of the cut-and-choose protocol at least one of the remaining ciphertexts is a correct randomization of $\overline{x_S}$ with high probability.

Let $\ell_\delta = \lceil \log p \rceil + \ell + \log k + 2$ be the bit-length of the randomizer such as δ. This size ensures that the k unopened $x_S + \delta_i$ values statistically hide x_S. Furthermore, we require that $N > 3 \cdot 2^{\ell_\delta}$ to ensure no overflows occur.

In our security proofs, see Section 5.6, we show that an adversary cannot learn anything useful about x_S despite seeing $\overline{x_S}$ and having access to the TS. We reduce to the CPA security of the homomorphic encryption scheme to show that an adversary cannot use $\overline{x_S}$ to learn something about x_S. However, in the reduction to CPA security, we cannot decrypt ciphertexts. Yet, in the GenShares protocol, the TS *must* decrypt a randomized version of $\overline{x_S}$ to recover the randomized key share. To allow us to correctly answer GenShares queries *without* decrypting ciphertexts, the user additionally creates a commitment Δ_i to δ_i and κ_i. In our proof of security, we use the extractability of $\mathsf{ExtCommit}(\cdot, \cdot)$ to extract δ_i from these commitments, thus allowing us to answer GenShares queries without actually decrypting.

Using an additively homomorphic CCA2 secure encryption scheme would obviate the need for the extractable commitments Δ_i, simplifying the scheme. Unfortunately, to the best of our knowledge no additively homomorphic CCA2 secure scheme exists. The RCCA scheme by Canetti et al. [CKN03] is not homomorphic, the schemes by Prabhakaran and Rosulek [PR08] are multiplicatively homomorphic, and the fully homomorphic scheme by Lai et al. [Lai+16] is not CCA2 secure.

PROTOCOL 2. The ObtainKeyShareToken protocol is run between a user and the TS.

1. The user recovers $(x_U, \overline{x_S}, pk_{id})$ from storage, and authenticates to the TS. The TS aborts if this user exceeded the rate-limit for the current epoch, was banned, or was blocked. Otherwise, the TS looks up the user's record $(x_S, \overline{x_S}, pk_{id})$.

2. The TS randomly chooses a subset $\mathcal{D} \subset \{1, \ldots, 2k\}$ of cardinality k of indices of ciphertexts that it will check at step 5. The TS commits to $\mathcal{D}$ by picking

$\theta \in_R \{0,1\}^{2\ell}$ and sending $\Delta = \mathsf{ExtCommit}(\mathcal{D}, \theta)$ to the user.

3. The user picks randomizers $\delta_1, \ldots, \delta_{2k} \in \{0,1\}^{\ell_\delta}$ to randomize the encrypted secret share $\overline{x_S}$, randomizers $\kappa_1, \ldots, \kappa_{2k} \in \mathcal{R}$ to create ciphertexts, and randomizers $r_1, \ldots, r_{2k} \in \mathbb{Z}_p$ and $\xi_1, \ldots, \xi_{2k} \in \{0,1\}^{2\ell}$ for the commitments and sets:

$$\begin{aligned} c_i &= \overline{x_S} \cdot \mathbf{E}^+_{pk}(\delta_i; \kappa_i) \\ C_i &= \mathsf{Commit}(H(c_i), r_i) \\ \Delta_i &= \mathsf{ExtCommit}((\delta_i, \kappa_i), \xi_i), \end{aligned} \tag{3}$$

for $i = 1, \ldots, 2k$. Finally, she sends the commitments $C_1, \ldots, C_{2k}$ and $\Delta_1, \ldots, \Delta_{2k}$ to the TS. Note that the commitments C_i and Δ_i are computationally binding and hiding.

4. The TS opens the commitment Δ by sending the subset $\mathcal{D}$ and the randomizer θ to the user. The user checks that $\Delta = \mathsf{ExtCommit}(\mathcal{D}, \theta)$, and aborts if the check fails.

5. The user opens the requested commitments by sending $(c_i, \delta_i, \kappa_i, r_i, \xi_i)_{i \in \mathcal{D}}$ to the TS. The TS checks that all disclosed values are constructed as per equation (3) and that $\delta_i < 2^{\ell_\delta}$. If any check fails, the TS bans the user.

6. Next, the user generates a token identifier $id \in_R \mathbb{Z}_p$ at random. Let $\mathcal{H} = \{i_1, \ldots, i_k\} = \{1, \ldots, 2k\} \setminus \mathcal{D}$ be the set of indices of unopened commitments. For the blind signature the user picks $r \in_R \mathbb{Z}_p$ and creates a commitment

$$C = \mathsf{Commit}((id, \epsilon, H(c_{i_1}), \ldots, H(c_{i_k})), r)$$

to the unopened ciphertexts, the epoch ϵ and id. Then, she encrypts the token identifier id as $\overline{id} = \mathbf{E}_{pk_{id}}(id)$, and sends C and $\overline{id}$ to the TS. Finally, she proves in zero-knowledge to the TS that $\overline{id}$ encrypts the token identifier id in C against her own public key pk_{id} and that C commits to the unopened ciphertexts, i.e.,

$$\begin{aligned} PK\{((c_i, r_i, \eta_i)_{i \in \mathcal{H}}, id, r) \;:\; & \overline{id} = \mathbf{E}_{pk_{id}}(id) \wedge \\ & \forall i \in \mathcal{H}\, [C_i = \mathsf{Commit}(\eta_i, r_i)] \wedge \\ & C = \mathsf{Commit}((id, \epsilon, \eta_{i_1}, \ldots, \eta_{i_k}), r)\}, \end{aligned}$$

where $\eta_i = H(c_i)$. The TS checks this proof.

7. If any check fails, the TS bans the user and aborts the protocol. If all checks pass, the TS runs a blind signature protocol with the user on the commitment C so that the user obtains a signature σ on the tuple $(id, \epsilon, H(c_{i_1}), \ldots, H(c_{i_k}))$. The user stores $\tau = (\sigma, \epsilon, id, (c_i, \kappa_i, \delta_i)_{i \in \mathcal{H}})$. The TS stores $\overline{id}$.

The following lemma states that even if a user is malicious, at least one of the ciphertexts c_i must be correctly formed. (See Appendix B.1 for the proof.)

Lemma 1. Consider a token $\tau = (\sigma, \epsilon, id, (c_i, \kappa_i, \delta_i)_{i=1,\ldots,k})$ obtained using the above protocol by a (potentially malicious) user U with corresponding TS key-share x_S. Let $\Delta_1, \ldots, \Delta_k$ be the corresponding set of commitments used during the obtain step. Then, with probability $1 - 1/\binom{2k}{k}$ there exists an index i^*, and randomizers $\delta^* < 2^{\ell_\delta}$, κ^*, and ξ^* such that:

$$\begin{aligned} c_{i^*} &= \mathbf{E}^+_{pk}(x_S + \delta^*; \kappa^*) \\ \Delta_{i^*} &= \mathsf{ExtCommit}((\delta^*, \kappa^*), \xi^*). \end{aligned}$$

Using a Key-share Token. When using a token, the user sends the tuple $(id, \epsilon, c_{i_1}, \ldots, c_{i_k})$ and the signature σ to the TS, and provides an index j of the ciphertext c_j that the TS should decrypt. The TS uses the corresponding plaintext as the key in the threshold-cryptographic protocol. We know from Lemma 1 that at least one index i^* exists such that c_{i^*} is correctly formed. Key-share tokens resemble Chaum et al.'s e-cash tokens [CFN]. For the e-cash tokens it suffices if some indices are correct, in TANDEM, however, the user chooses the index j, and we must thus ensure that c_j in particular is correct. To enable the TS to check this, the user also reveals the differences $\gamma_i = \delta_j - \delta_i$ for all $i = 1, \ldots, k$. If these differences are correct then because c_{i^*} is a randomization of x_S, so must be c_j.

PROTOCOL 3. The GenShares protocol is run between an anonymous user and the TS.

1. The user a token $\tau = (\sigma, \epsilon, id, (c_i, \kappa_i, \delta_i)_{i=1,\ldots,k})$ as input and connects to the TS via an anonymous channel. She sends $(id, \epsilon, c_1, \ldots, c_k)$ and the blind signature σ.

2. Next, the user finds j such that $\delta_j \geq \delta_i$ for all i and computes $\gamma_i = \delta_j - \delta_i \geq 0$ and $\nu_i = \kappa_j \cdot \kappa_i^{-1}$ such that

$$c_j = c_i \cdot \mathbf{E}^+_{pk}(\gamma_i; \nu_i) \tag{4}$$

   for $i = 1, \ldots, k$. Finally she sends j, and $\gamma_1, \ldots, \gamma_k, \nu_1, \ldots, \nu_k$ to the TS.

3. The TS verifies that the γ_is and ν_is satisfy equation (4), that σ is a correct signature on $(id, \epsilon, H(c_1), \ldots, H(c_k))$, token id was not blocked, ϵ corresponds to the current epoch, and that $\gamma_i < 2^{\ell_\delta}$. The TS aborts if any check fails.

4. The TS decrypts c_j to compute $\tilde{x}_S = \mathbf{D}^+_{sk}(c_j) \pmod p$.

5. The user calculates her key share $\tilde{x}_U$ as:

$$\tilde{x}_U \equiv x_U - \delta_j \pmod p$$

Using Lemma 1, we can show that the decrypted element c_j must also be of the right form. (See Appendix B.1 for the proof.)

Lemma 2. If revealed token $(id, \epsilon, c_1, \ldots, c_k,)$ with j, $\gamma_1, \ldots, \gamma_k$ and $\nu_1, \ldots, \nu_k$ satisfies equation (4), then with probability $1 - 1/\binom{2k}{k}$ there exists $\delta < 2^{\ell_\delta + 1}$ such that

$$\mathbf{D}^+_{sk}(c_j) = x_S + \delta$$

where x_S is the TS key-share for the corresponding user.

The range proof in registration is essential. The range proof in equation (2) in the RegisterUser protocol ensures that the plaintext $x_S = \mathbf{D}^+_{sk}(\overline{x_S})$ is small compared to the randomizers δ_i. As a result, the randomized ciphertexts c_i statistically hide x_S. It is not sufficient to skip the range proof and instead choose the randomizers δ_i from the full plaintext domain $[N]$ to hide x_S. Without the range proof, the TS can construct tokens that it can later recognize by exploiting the fact that a large x_S results in a reduction modulo N. More precisely, the TS can set x_S of its target user somewhat large, so that $x_S + \delta_j > N$ (with a non-negligible probability). The user believes that the TS derives $x_S + \delta_j \pmod p$ (because she believes no modular reduction took place) and compensates accordingly. However, the TS actually derives $\tilde{x}_S = (x_S + \delta_j \mod N) \pmod p = x_S + \delta_j - (N \mod p)$. To test if the current token is from its target user, the TS adds $(N \mod p)$ to $\tilde{x}_S$. If the guess was correct, the

TCP completes correctly, otherwise the protocol fails. This allows the TS to detect specific users.

Blocking the Key. To block her key, the user runs the BlockShare protocol with TS to ensure no new key-share tokens are created for her, and that all her unspent tokens are blocked.

Protocol 4. The BlockShare protocol is run by a user and the TS. The user takes as input her long-term key sk_{id} (which she recorded outside her device). The user authenticates to the TS (possibly using sk_{id}). The TS marks the user as blocked, so that it will no longer issue new tokens. Then they continue as follows to invalidate unspent tokens. The TS sends a list of all encrypted token identifiers $\overline{id}_1, \ldots, \overline{id}_t$ that the user obtained in this epoch. The user looks up a list of all spent token identifiers (see below). The user then uses sk_{id} to decrypt $\overline{id}_1, \ldots, \overline{id}_t$ and sends the decrypted token identifiers that have not yet been spent to the TS. The TS will then block all tokens with these identifiers.

Since we assume the TS is honest with respect to blocking, the TS accurately provides the list of encrypted token identifiers. In the ObtainKeyShareToken protocol, the user verifiably encrypts the token identifier id. As a result, even if the user's device is corrupted, the TS stores a correct encryption $\overline{id}$ of id, so the above procedure blocks all unspent tokens.

In the unlikely case that a user cannot recover the identifiers, the attacker can nevertheless use the TS only a limited number of times, as the attacker is still subject to the rate-limit.

List of spent tokens. The TS is malicious with respect to privacy. So, it might try to trick the user into revealing the identifiers of tokens she has already spent (thus revealing that these tokens were hers). In particular, the TS is not trusted to provide an accurate list of spent tokens. Therefore, we propose that users externally store spent token identifiers, so that they have a reliable record. Alternatively, the TS can keep a verifiable log of spent tokens by appending spent token identifiers to a public append-only log (users must then verify that each spent token identifier is in fact added to the log). Users then use this log as a record of spent tokens. Finally, if epochs are short, and users are willing to risk revealing their actions in the current epoch, they can also use a list provided by the TS. If the TS cheats, users reveal at most their actions within the most recent epoch when they block their keys.

5.5.2 Alternative constructions

An alternative method to construct tokens could be to use an authenticated encryption scheme that the user and the TS evaluate using secure multi-party computation [Yao86]. The server inputs its key share x_S while the user inputs the randomizer δ. The user's output is the authenticated encryption of $x_S + \delta$ for the TS's symmetric key which serves as token. To ensure that the TS cannot recognize this token, the protocol should resist malicious servers and the circuit should validate the TS's input (i.e., that they are always the same). Even though recent secure two-party computation protocols that are secure against a malicious server boast impressive performance [WRK17], they still require at least one order of magnitude more computational power as well as more bandwidth than our custom scheme.

Another simple alternative construction is to let users retrieve $\overline{x_S} = \mathsf{Enc}(x_S)$ using private information retrieval (PIR) via an anonymous channel—the user must still hide her identity. Then, users randomize $\overline{x_S}$ similarly to our construction, and the

TS decrypts the ciphertext to recover $\overline{x_S} + \delta$, which it then uses in the TCP. To enable blocking of keys, the TS needs to frequently refresh its encryption keys, effectively invalidating previously retrieved ciphertexts $\overline{x_S}$. This simple protocol, however, has serious drawbacks. First, blocking is only enforced upon key refreshing, thus the timespan when compromised keys can be used depends on the refreshing schedule of the TS. Second, because the encryption of x_S for the current period can be randomized as often as the user wants (and the use of PIR precludes record-keeping), this scheme cannot provide rate-limiting. Third, because the TS acts as a decryption oracle for a homomorphic encryption scheme, which is only CPA secure, proving security in this setting requires very strong and non-standard assumptions.

5.6 Security and Privacy of Tandem

In this section we formalize the security and privacy properties offered by Tandem. We refer to the appendix for the complete security and privacy proofs.

5.6.1 Security of Tandem

We capture the security of Tandem using the following game. It models that if the user's key is compromised (e.g., her device is stolen), the user can block the use of her key, provided that the Tandem server remains honest.

Game 1. The *Tandem security game* is between a challenger controlling the TS and the SP, and an adversary controlling up to n users. The adversary's goal is to complete a threshold-cryptographic protocol for a blocked user.

Setup phase The challenger sets up the TS and the SP. The challenger runs RegisterUser with the adversary for each of the n users the adversary controls.

Query phase During the query phase, the adversary can ask the TS to run the RegisterUser, ObtainKeyShareToken and BlockShare protocols with users controlled by the adversary. Moreover, the adversary can make RunTCP queries to the challenger. In response, the TS first runs the GenShares protocol with the user (controlled by the adversary), followed by a run of the TCP protocol.

Selection phase At some point the adversary outputs the identifier of a blocked user U^* on which it wants to be challenged later. To allow the challenger to confirm that all unspent tokens are blocked (to prevent trivial wins), the adversary also outputs the long term secret sk_{id} of user U^*. The challenger checks sk_{id} against the recorded public key pk_{id} and then blocks all tokens of user U^* using sk_{id}. The adversary loses if sk_{id} is not correct.

Second query phase The adversary can keep asking the TS to run RegisterUser, ObtainKeyShareToken, BlockShare protocols. The adversary can also make RunTCP queries as before (however, following the protocols the TS will not allow ObtainKeyShareToken queries of blocked users).

Challenge phase Finally, upon request of the adversary, the challenger acts as SP in the TCP protocol. At the same time, the adversary may still make queries and run protocols as in the previous phase. The adversary wins if it successfully completes the TCP with the SP on behalf of the blocked user U^*. To

prevent trivial wins, this TCP protocol must be completable only by user U^*.[13]

In this game, all users are automatically corrupted right from the moment they start the registration protocol. This models the notion that users can even be blocked if the adversary is present right from the start, and also implies that honest users—which are only corrupted later—can still be blocked.

Game 2. The *Tandem rate-limiting* game is identical to the Tandem security game, except that in the selection phase the adversary outputs a rate-limited user (i.e., a user who is not allowed to obtain more tokens in this epoch).

Of course, to have security using Tandem, the TCP itself must be secure. Hence, we require that even if a malicious user has interacted many times with the TS, she cannot use her key when she does not have access to the TS. We formalize this using the following game.

Game 3. The *TCP security game* is between a challenger controlling the TS and the SP and the adversary controlling a malicious user.

Setup phase During the setup phase, the adversary generates $x_U \in_R \mathbb{Z}_p$, whereas the TS, controlled by the challenger, generates $x_S \in_R \mathbb{Z}_p$.

Query phase In the query phase, the adversary can make TCP(δ) queries to request that the TS runs TCP.TS($x_S + \delta$) with the user. The adversary is responsible for running TCP.U. Optionally, the adversary-controlled user can communicate with the challenger-controlled SP running P() as well.

Challenge phase In the challenge phase, the adversary is not allowed to make TCP queries. Instead, it interacts solely with the challenger-controlled SP running P(). The adversary wins if the SP accepts.

Theorem 1. No PPT adversary can win the *Tandem security game* or the *Tandem rate-limiting game* with non-negligible probability, provided that the TCP is secure (i.e., no PPT adversary can win the TCP security game).

Proof sketch. We prove the security of the scheme by reducing it to the TCP security property. First, we show how to run GenShares without decrypting ciphertexts. During the ObtainKeyShareToken protocol, we model hash functions as random oracles to allow us to extract the token identifier *id* from the proof of knowledge in step 6, and the unopened randomizers δ_i from the extractable commitments Δ_i in step 3. Hence, during GenShares we can identify the user, and thus the corresponding key share x_S, as well as the randomizer δ_j (with overwhelming probability, using Lemma 2).

Knowing x_S and δ_j we no longer need to decrypt ciphertexts to run GenShares, therefore, we can use the CPA security of the homomorphic encryption scheme to replace the initial ciphertext $\overline{x_S} = \mathbf{E}^+_{pk}(x_S)$ for the challenge user by $\overline{x_S} = \mathbf{E}^+_{pk}(0)$ an encryption of 0. During GenShares we add x_S to compensate. (To enable the reduction to CPA, we simulate the range proof in step 2 of RegisterUser.)

Finally, we answer all queries for the challenge user using the TCP security oracle. Hence, a break of the Tandem security game results in a break of the TCP security game. □

See Appendix B.3 for the full proof.

[13] One option is that the TCP protocol identifies the user. So for example, for the attribute-based credential TCPs, this means that the showing protocol must disclose an attribute that identifies U^*. Another option is to make sure that user U^* is the only user who can successfully complete the protocol, e.g., by revoking all other credentials.

5.6.2 Privacy of Tandem

The following game models that Tandem provides privacy for users. A malicious Tandem server cannot distinguish between two honest users performing a transaction using the Tandem server even if it colludes with the service provider, provided that the service provider on its own cannot distinguish transactions by these two honest users. The following privacy game asks the TS to recognize users for which it earlier issued a key-share token.

Game 4. The *Tandem privacy game with colluding SP* is between a challenger, who controls two honest users U_0 and U_1, and an adversary $\mathcal{A}$ who controls the TS and the SP.

Setup phase The adversary $\mathcal{A}$ outputs the number of key-share tokens n_T each honest user should obtain. The adversary is responsible for setting up the SP and the TS, i.e., it should publish a public key pk. Next, the honest users U_0 and U_1 interact with the adversary-controlled TS to obtain n_T key-share tokens each. First, U_0 runs ObtainKeyShareToken n_T times to obtain tokens $\tau_{0,1}, \ldots, \tau_{0,n_T}$. Then, U_1 runs the obtain protocol n_T times to obtain tokens $\tau_{1,1}, \ldots, \tau_{1,n_T}$.

Query phase During the query phase, the adversary can make $\mathsf{RunTCP}(U_i, j, \mathsf{in}_U)$ queries to request that user U_i uses token $\tau_{i,j}$ and then runs the TCP with input in_U. If $i \in \{0,1\}$ and user U_i did not use token $\tau_{i,j}$ before, then user U_i, controlled by the challenger, first runs GenShares with the TS using token $\tau_{i,j}$ and then runs $\mathsf{TCP.U}(\mathsf{in}_U)$ with the TS and the SP (running TCP.TS and P respectively).

Challenge phase At some point, the adversary outputs a pair of token indices (i_0, i_1) for user U_0 and U_1 respectively on which it wants to be challenged. Let $\tau_0 = \tau_{0,i_0}$ and $\tau_1 = \tau_{1,i_1}$ be the corresponding tokens. The adversary loses if either token τ_0 or τ_1 has been used before. Then, the challenger picks a bit $b \in \{0,1\}$ and proceeds as if the adversary first made a $\mathsf{RunTCP}(U_b, \tau_b)$ query and then a $\mathsf{RunTCP}(U_{1-b}, \tau_{1-b})$ query.

Guess phase The adversary outputs a guess b' of b. The adversary wins if $b' = b$.

The privacy game models the fact that there is *no* time correlation between when tokens are obtained by a user, and when they are spent by a user. At the same time, the adversary has full control over the TS and the SP, so this game also models the fact that the TS and the SP *can* correlate events that they see.

Since the SP is controlled by the adversary, the TCP must ensure privacy with respect to the SP and the TS, if all that the TS sees are randomized secret shares. We formalize this in the following game.

Game 5. The *TCP privacy game with colluding SP* is between a challenger controlling two honest users U_0 and U_1 and an adversary $\mathcal{A}$, controlling the TS and the SP.

Setup The adversary publishes the TS public key and is responsible for setting up the SP. The challenger sets up its users. First, user U_0 generates $x_{0,U} \in_R \mathbb{Z}_p$ while the TS generates $x_{0,S} \in_R \mathbb{Z}_p$, then U_1 and TS similarly generate $x_{1,U}$ and $x_{1,S}$. Finally, the TS sends $x_{0,S}$ and $x_{1,S}$ to users U_0 and U_1 respectively.

Queries Adversary $\mathcal{A}$ can make $\mathsf{RunTCP}(i, \mathsf{in}_U)$ queries, to request U_i to run the TCP protocol using input in_U with the TS and the SP (both controlled by $\mathcal{A}$). In the first step, the user picks $\delta \in_R \mathbb{Z}_p$ and sends the randomized secret-share

$\tilde{x}_S = x_{i,S} + \delta \pmod p$ to the TS. The user then sets $\tilde{x}_U = x_{i,U} - \delta$ and runs $\mathsf{TCP.U}(\tilde{x}_U, \mathsf{in}_U)$ with the TS and the SP running TCP.TS and P respectively.

Challenge Adversary $\mathcal{A}$ outputs an input in_U. Challenger picks a bit $b \in_R \{0,1\}$. Then the challenger acts as if $\mathcal{A}$ first made a $\mathsf{RunTCP}(b, \mathsf{in}_U)$ query, and then a $\mathsf{RunTCP}(1-b, \mathsf{in}_U)$ query.

Guess The adversary outputs a guess b' for b, $\mathcal{A}$ wins if $b = b'$.

Tandem also offers privacy for users against the TS alone. That is, even if the service provider can identify users, the TS cannot observe their key-usage patterns as long it does not collude with the service provider. (If the SP can identify users, then so can a coalition of the TS and the SP, so we exclude this case to prevent a trivial win.) We model this situation as a variant of the previous two games.

Definition 2. The *Tandem privacy game with honest SP* and the *TCP privacy game with honest SP* are as in Game 4 and Game 5 above, however, the challenger controls the SP. The adversary can interact with the SP as a normal user.

Theorem 2. No PPT adversary can win the *Tandem privacy game with colluding SP* (respectively the *Tandem privacy game with honest SP*) with probability non-negligibly better than 1/2, provided that the TCP is privacy-friendly (i.e., no PPT adversary can win the TCP privacy game with colluding SP respectively the TCP privacy game with honest SP).

Proof sketch. We first argue that we can remove all identifying information from the key-share tokens of the challenge users. First, we extract the server's key-shares $x_{0,S}$ and $x_{1,S}$ from the proof of knowledge in step 2 of the RegisterUser protocol. Then we simulate the proof of knowledge in step 6 of ObtainKeyShareToken, replace the ciphertext $\overline{id}$ by the encryption of zero (using the CPA security of the ElGamal encryption scheme), extract the subset $\mathcal{D}$ so that we can send random commitments C_i, Δ_i for $i \notin \mathcal{D}$ (because the commitment schemes are computationally hiding), and finally, we set the unrevealed ciphertexts $c_i = \mathbf{E}^+_{pk}(\delta_i, \kappa_i)$ for $i \notin \mathcal{D}$. None of these changes are detectable by the adversary during ObtainKeyShareToken.

To use tokens as requested by the adversary, the challenge users add $x_{i,S}$ to their long-term share x_U to compensate for the changes made so that GenShares completes successfully. Moreover, because the randomizers δ_i statistically hide x_S, the adversary cannot detect the final change to the ciphertexts during GenShares.

Therefore, by the blindness of the signature scheme, we can swap tokens between the users and still simulate protocols perfectly. Therefore, any adversary that can then still distinguish users must break the security of the TCP privacy game (with a colluding SP or with a honest SP). To extract secrets from the proofs and commitments, and to make the final reduction to TCP privacy, we model hash functions as random oracles. □

See Appendix B.4 for the full proof.

5.7 Securing protocols with Tandem

Recall that to use Tandem to protect the private key in a cryptographic scheme we must convert the protocols into linearly randomizable threshold-cryptographic protocols.

TS		User		Issuer
$\tilde{x}_S \in \mathbb{Z}_p, B_0$		$\tilde{x}_U \in \mathbb{Z}_p, s', g, B, B_0, B_1$		sk_I
$\hat{x}_S \in_R \mathbb{Z}_p$		$\hat{x}_U, \hat{s}' \in_R \mathbb{Z}_p$		
$u_S = B_0^{\hat{x}_S}$	$\xrightarrow{u_S}$	$u_U = B_0^{\hat{x}_U}$		
		$\tilde{U} = B^{\hat{s}'} \cdot u_U \cdot u_S$	$\xrightarrow{\tilde{U}}$	
	$\xleftarrow{c}$		$\xleftarrow{c}$	$c \in_R \mathbb{Z}_p$
$r_S = \hat{x}_S + c \cdot \tilde{x}_S$	$\xrightarrow{r_S}$	$r_{s'} = \hat{s}' + c \cdot s'$		
		$r_U = \hat{x}_U + c \cdot \tilde{x}_U$		
		$r = r_U + r_S$	$\xrightarrow{r_{s'}, r}$	$\tilde{U} \stackrel{?}{=} U^{-c} \cdot B^{r_{s'}} \cdot B_0^r$

Figure 16: Full details of the proof of knowledge of the user's commitment $U = B^{s'} B_0^{x_U} B_0^{x_S}$ in the BBS+ TCP issuance protocol. The TANDEM server only knows $\tilde{x}_S$ and the user knows $\tilde{x}_U$ and the randomness s' (recall $\tilde{x}_S$ and $\tilde{x}_U$ are the respective outputs of the GenShares protocol). The TS effectively creates a zero-knowledge proof of knowing $\tilde{x}_S$.

For this composition to be secure, the threshold-cryptographic protocols must satisfy the natural security definition (see Game 3). For this composition to be private, i.e., so that the TANDEM server alone respectively by colluding with the SP cannot identify the user, the threshold-cryptographic protocols must additionally be private (see Game 5).

Many traditional threshold-cryptographic schemes already satisfy these requirements. Threshold variants of Schnorr [Gen+07] and RSA signatures [Sho00] as well as ElGamal-based [ElG84; SG02] and RSA encryption [Sho00] schemes rely on Shamir secret-sharing and are thus linearly randomizable. Moreover, the threshold protocols are private, i.e., the server-side protocols for signing and decrypting, respectively, operate solely on the secret-share and the common input, the message or ciphertext.

Threshold-cryptographic versions of electronic cash schemes [CHL05; Mie+13] and attribute-based credential (ABC) schemes [ASM06; Bra00; CH02; CL02] can also be used with TANDEM. For some of these, the threshold-cryptographic versions already exist [Bra00]. For the others, the threshold-cryptographic versions of the zero-knowledge proofs on which these schemes are based must be created. As an example, we now show how convert the BBS+ ABC scheme [ASM06] into a TANDEM-suitable threshold-cryptographic scheme.

5.7.1 The use-case of ABCs

Attribute-based credentials can be conceptualized as digital equivalents to classic documents like passports, driver's license, student cards, etc. The owner of a credential can selectively disclose any subset of attributes to a service provider in such a way that the the validity of the disclosed attributes can be validated. In many ABC systems credentials are unlinkable, that is, users are anonymous within the set of users having the same disclosed attributes.

To bind credentials to a user, and to ensure that only the owner can operate with them, credentials contain the user's secret key. Typically, all credentials of a user contain the same secret key. When credentials are stored on insecure platforms such as smart phones or personal computers TANDEM can be used to strengthen the

security of the secret key. This ensures that valuable credentials cannot be abused, and can be blocked, while preserving users' privacy.

To use ABCs with TANDEM we need to convert the protocols for issuing and verifying credentials into threshold-cryptographic alternatives that are secure, private, and linearly randomizable. During issuance, the issuer (taking the place of the service provider in Section 5.3) provides the user with a credential bound to the user's secret key. The issuer does not learn the user's secret key. During verification, a user authenticates to a service provider by selectively disclosing attributes from her ABCs.

In typical ABC schemes, these two protocols rely heavily on zero-knowledge proofs over the user's secret key. In the remainder of this section, we show how these non-threshold protocols for BBS+ credentials [ASM06] can be converted to threshold-cryptographic versions suitable for TANDEM.

BBS+ credentials are anonymous credentials built from BBS+ signatures [ASM06]. BBS+ signatures operate in a pairing setting and rely on discrete-logarithm based assumptions. Let $(\mathbb{G}_1, \mathbb{G}_2)$ be a bilinear group pair, both of prime order p, generated by g and h respectively. The pairing is given by $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ where $\mathbb{G}_T$, also of order p, is generated by $\hat{e}(g, h)$. Let l be the number of attributes. In the BBS+ credential scheme, an issuer randomly chooses generators $B, B_0, .., B_l \in_R \mathbb{G}_1$, picks a private key $sk_I \in_R \mathbb{Z}_p$, and computes $w = h^{sk_I}$. The issuer's public key is $pk_I = (w, B, B_0, .., B_l)$.

Obtaining a credential. Attribute-based credentials contain the user's secret key as an attribute. For simplicity, we describe the TANDEM BBS+ issuance and showing protocols below with two attributes: the secret key x and an issuer-determined attribute a_1. To obtain a credential, the user (and the TANDEM server) run the following TCP version of the issuance protocol with the issuer. The issuance protocol is run jointly by the user, TS, and an issuer. Let $\tilde{x}_U$ and $\tilde{x}_S$ be the two shares of the user's secret key $x = \tilde{x}_U + \tilde{x}_S$ that are held by the user and the TS respectively after running GenShares. The user first commits to her secret key x, to allow the issuer to blindly sign it. As we share the user's secret key between the user and the TS, they both have to participate in creating the commitment. First, the user sends B_0 to the TS so that it can compute $B_0^{\tilde{x}_S}$ before sending back to the user. Then the user and the TS create a commitment $U = B^{s'} B_0^{\tilde{x}_U} B_0^{\tilde{x}_S} = B^{s'} B_0^{x}$ where $s' \in_R \mathbb{Z}_p$. To prove to the issuer that U is well-formed, the user and the TS construct the proof

$$PK\{(x, s') : U = B^{s'} B_0^{x}\}. \tag{5}$$

Fig. 16 shows how to construct this proof. If this proof of knowledge verifies, the issuer randomly generates $s'', e \in_R \mathbb{Z}_p$ and calculates

$$A = \left(gB^{s''} U B_1^{a_1}\right)^{\frac{1}{e+sk_I}} \quad \in \mathbb{G}_1$$

and the tuple (A, e, s'') to the user. The user calculates $s = s' + s''$ and stores the credential $\sigma = (A, e, s)$.

Showing a credential. After the issuance protocol, the user can show the credential to a service provider to get access to a service or a resource. Again we convert the showing protocol into a TCP that uses the TANDEM server.

In the showing protocol, the user proves the possession of a credential $\sigma = (A, e, s)$ over her key x and the attribute a_1. To show she possesses such a credential, while

hiding her key and disclosing her attribute, she proves in zero-knowledge that

$$\hat{e}(A, h^e w) = \hat{e}(gB^s B_0^x B_1^{a_1}, h). \tag{6}$$

To prove the validity of this equation in zero-knowledge, without revealing any of the values A, e, s (that would make the user linkable), we follow the approach by Au et al. [ASM06]. Let g_1, g_2 be two extra generators in $\mathbb{G}_1$. First, the user creates a commitment $C_1 = Ag_2^{r_1}$ to A, where r_1 is a randomizer chosen from $\mathbb{Z}_p$ by the user. The user then commits to her randomizer as well using $C_2 = g_1^{r_1} g_2^{r_2}$ where $r_2 \in_R \mathbb{Z}_p$. The user sends these commitments to the service provider. These commitments perfectly hide the value of A. Finally, she and the TS engage in the following zero-knowledge proof with the service provider:

$$PK\Big\{(r_1, r_2, \alpha_1, \alpha_2, e, x, s) : C_2 = g_1^{r_1} g_2^{r_2} \wedge C_2^e = g_1^{\alpha_1} g_2^{\alpha_2} \wedge \hat{e}(C_1, w) \cdot \hat{e}(C_1, h)^e = \hat{e}(g, h)\hat{e}(B, h)^s \hat{\mathbf{e}}(\mathbf{B_0}, \mathbf{h})^x \cdot \hat{e}(B_1, h)^{a_1} \hat{e}(g_2, w)^{r_1} \hat{e}(g_2, h)^{\alpha_1}\Big\}$$

to prove that she indeed posseses the signature over the hidden and the disclosed attributes and that equation (6) is satisfied. In the proof, $\alpha_1 = er_1$ and $\alpha_2 = er_2$. The user can easily generate the proofs for the first two conjuncts. Only the third conjuct contains the user's secret key x of which the user only has a secret share. Thus, the user has to contact the TS to construct this part of the proof. This proof is just a proof of representation, like in equation 5, albeit a bit more complex. As a result, a very similar construction as in Fig. 16 allows the user and the TS to jointly compute this proof.

Security and privacy of the TCPs. These TCPs satisfy the TCP security and privacy notions defined in Section 5.6. For security (see Game 3), note that the TS computes zero-knowledge proofs of knowing $\tilde{x}_S$. A malicious user learns nothing about $\tilde{x}_S$ (thus nor x_S) as a result of the zero-knowledge property. Hence, the TCP showing and issuance protocols satisfy the TCP security property.

For privacy (see Game 5), the TS operates on a fully randomized key $\tilde{x}_S$, so the TS cannot distinguish users based on the key if the SP is honest. The indistinguishability property of the credential scheme guarantees that the TS cannot distinguish users based on the resulting showing proof by colluding with the SP either. Thus, the TCP showing protocol is private for both honest and colluding SPs.

5.7.2 Rate-limiting in ABCs

Anonymous users can use the cover of privacy to misbehave, negatively impacting the system. ABC systems are not exempt from such misbehavior. Suppose, for example, that a user shares her "I am older than 18" credential with many under-aged users who do not hold such a credential. Then, those under-aged users can incorrectly convince service providers that they are over 18 years of age. If this happens often, service providers can no longer rely on these credentials to verify that a user is older than 18.

To limit such misbehavior, ABCs could benefit from rate-limiting. One method to limit abuse is to rate-limit credentials by ensuring that credentials can only be used a limited number of times. For instance, solutions such as n-times anonymous credentials [Cam+06] use custom cryptographic techniques to construct a special type of ABC that can be used only a limited number of times.

Tandem can achieve a similar type of rate-limiting *without* modifying the underlying cryptographic construction of ABCs. To rate-limit use of a system, the TS enforces a per-user and per-epoch limit q on the number of tokens it issues per user and per epoch. As a result, no credential can be shown more than q times per epoch. This approach limits *all* credentials associated to a user's key. If desired, Tandem can equally be applied on a per-credential basis.

This rate-limiting strategy *requires* that all users use Tandem. However, recall that the SPs (issuers and verifiers) cannot detect the use of Tandem, allowing users to forego sharing their keys with the TS, thus avoiding the rate limit. To enable the TS to enforce a rate-limit on all credentials, issuers must only issue credentials on keys that are shared with the TS.

A small change to the threshold-cryptographic version of the issuance protocol enables the issuer to confirm that users use Tandem. To signal its involvement, the TS signs its proof (u_S, c, r_S) and sends the signature σ to the user. The user forwards the messages u_S, r_S and σ from the TS to the issuer together with its own messages u_U, $r_{s'}$ and r_U. The issuer, rather than the user, combines the proofs and verifies them. Moreover, the issuer checks the signature σ. If the signature and proofs are correct, then the user's key was shared with the TS and the issuer signs the credential.

5.8 Performance Evaluation

In this section, we evaluate Tandem's computational and bandwidth cost. We use an attribute-based credential instantiation as a case study and compare the performance when using no key protection, traditional threshold-cryptographic schemes, and Tandem.

Tandem consists of four protocols: RegisterUser, ObtainKeyShareToken, GenShares, and BlockShare. We implemented in C the time-critical protocols, ObtainKeyShareToken and GenShares.[14] We used Pedersen commitments [Ped91] as commitment scheme, and BBS+ credentials [ASM06] to construct the blind signature. We use the RELIC cryptographic library to implement them [AG].[15] We use a recent implementation [BCF17] of Joye and Libert's additive homomorphic encryption scheme [JL13] for our protocols. We set the modulus size to 2048 bits and the plaintext space to 394 bits, such that $N > 3 \cdot 2^{\ell\delta}$ for $k \leq 85$. With this setting, encrypting a single 394 bits plaintext takes 0.9 ms whereas it takes 24.2 ms to decrypt a ciphertext. We also experimented with an optimized implementation[16] of Paillier's encryption scheme [Pai99], but our experiments show that Joye and Libert's scheme gives better performance. Finally, we use ElGamal's encryption scheme [ElG85] to encrypt token identifiers.

We empirically measure performance on a single core of an Intel i7-7700 running at 3.6 GHz. Smart phones and tablets generally have slower processors. Yet, we believe that given our measurements, Tandem's performance would be practical on these devices.

Obtaining a token. We first justify our choices for the value of the security parameters k in our experiments. Our security analysis shows that an attacker can break Tandem's security property by constructing a key-share token for a blocked user

[14] The code will be available upon publication.

[15] We set up RELIC to use a BLS curve over a 381 bits field. This setup ensures 128 bits security, while the group order remains 255 bits.

[16] `https://github.com/mcornejo/libpaillier`

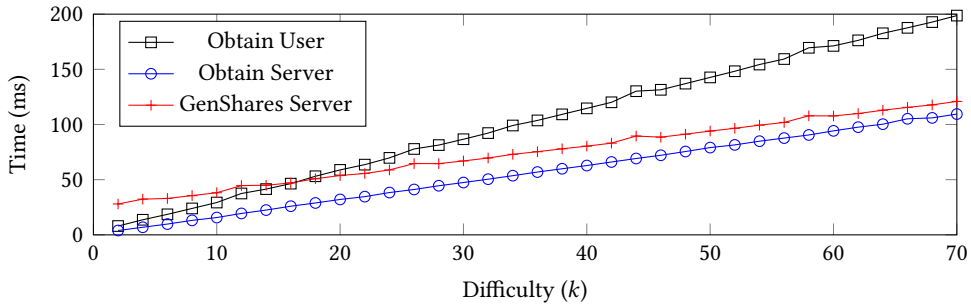


Figure 17: ObtainKeyShareToken protocol computing time at the user (black) and the server side (blue), and GenShares protocol computation time at the server side (red) for increasing difficulty levels k.

with probability $\binom{2k}{k}^{-1}$. Hence, $k = 42$ gives 80 bits of security, and $k = 66$ gives 128 bits security. However, ObtainKeyShareToken is an *interactive* protocol. The success probability of an attacker is limited by how often the TS lets the attacker try to construct a malicious token rather than by the adversary's computational power. Because the TS bans users trying to construct malicious tokens, one can choose a smaller k in practice. In a system with 100 000 users, $k = 20$ ensures that the probability that an attacker (corrupting all users) can at least *once* use any blocked key is less than 10^{-6}.

Fig. 17 shows the computing time (without communication) for the ObtainKeyShareToken protocol at the user (black) and server (blue) for different values of the parameter k. The homomorphic encryption scheme—creating the ciphertexts (user), and checking a subset of these (TS)—dominates the computational cost. Our experiments reveal that the timing variance accross executions is negligible. The bandwidth cost is low. For a security level of $k = 20$, the user sends about 26 KiB and receives less than 200 bytes.

Using the key. On the user side running GenShares is very cheap. Even for $k = 60$ the user requires less than 5 ms. In terms of bandwidth the user just needs to send 12 KiB for $k = 20$ and 36 KiB for $k = 60$. We show the server's computational cost for recovering the TS key-share from the token in Fig. 17. For a reasonable security level of $k = 20$, the server computational overhead is around 50 ms. The sending of the token in the GenShares protocol can be combined with the request to start the TCP resulting in no extra latency on top of the delay incurred by the Tor network [DMS04b] (1–2 s to send a small and receive amount of data on a fresh circuit[17]). Note that the circuit creation and GenShares can be run preemptively, thereby reducing the user-perceived delay.

Given the above measurements, a modern 4-core server can participate in approximately 50 TCPs per second (not counting the cost of the application-dependent TCP itself), i.e., serve 3 000 users per minute, requiring about 15 Mbit/s incoming bandwidth.

RegisterUser and BlockShare. These protocols are run few times (only upon registration and for blocking) and are thus not critical for scalability. We estimate the cost for RegisterUser to be well below a second for both the user and the TS (given its similarity with the ObtainKeyShareToken and GenShares protocols, and that the cost of the range proof is around 500 ms). In BlockShare the ElGamal decryption the

[17] As reported by `https://metrics.torproject.org/torperf.html`, visited July 6, 2018.

Table 7: Comparison of computational cost and properties between not using a TCP, using a traditional TCP, and using a TCP with Tandem ($k = 20$).

	No TCP	Traditional TCP	TCP + Tandem
Obtain Token			
User	-	-	59 ms
Server	-	-	32 ms
Run Protocol			
User	5 ms	5 ms	5 + 4 ms
Server	-	1 ms	1 + 54 ms
Key blocking	×	✓	✓
Rate limiting	×	✓	✓
Privacy	✓	×	✓

token identifiers dominates the run-time cost, we estimate it to be in the order of seconds for thousands of tokens.

Comparison. Table 7 compares the computational cost of creating a single BBS+ disclosure proof with 5 hidden attributes without key protection, using a traditional TCP, and using a Tandem-augmented TCP configured with $k = 20$.

Without a TCP, the credential showing is very fast and, as there is no party involved in the use of the key, the showing of the credential is perfectly private. However, it is not possible to perform key blocking nor limit the key-usage unless specific credentials (e.g. [Cam+06]) are used. When introducing a traditional TCP, the overhead is minimal (only 1 ms at the server side) and key blocking and rate limiting are possible, at the cost of privacy. Tandem provides the three properties. Without taking into account the ObtainKeyShareToken operation that happens offline, the user's overhead is negligible (4 ms), and well below a second (54 ms) for the server. In all cases, the cost of Tandem's cryptographic operations are very small compared to Tor's network cost.

5.9 Conclusion

Protecting cryptographic keys is imperative to maintain the security of cryptographic protocols. As users' devices are most of the time insecure, the community has turned to threshold-cryptographic protocols to strengthen the security of keys. When run with a central server, however, these protocols raise privacy concerns. In this paper, we have proposed Tandem, a provably secure scheme that, when composed with threshold-cryptographic protocols, provides privacy-preserving access to the keys. Tandem also enables users to block her keys and rate-limit their usage, in ways that previous work could not handle. Our proof-of-concept implementation of Tandem shows that for reasonable security parameters Tandem's protocols run in less than 60 ms, hence being suitable for use in practice.

Tandem is particularly suited for privacy-friendly applications such as eCash and ABCs because it retains their inherent privacy properties. Yet, Tandem can be used to strengthen a wide variety of primitives, including signature and encryption schemes, as long as they can be transformed into linearly-randomizable threshold protocols. Using attribute-based credentials we have shown that deriving such a threshold protocol can be done with standard techniques, and that thereafter adding

Tandem is straightforward.

6 Conclusion

In this deliverable we presented the technical developments regarding scalability of messaging protocols resulting from activities within WP4. As the scalability and content security of the NEXTLEAP secure messaging protocols was proven in other deliverables, this document focuses on results aimed at improving the anonymity and privacy of communications and key-related operations. We have produced two anonymous communications modules that complement the Tor network to 1) make it more robust against end-to-end correlation attacks, and 2) make it more usable from the point of view of the service provider. For key management we have improved the design of Claimchains to include protection against a new attack. We have also developed a protocol that enables users to enjoy the advantages of threshold threshold cryptography to improve key security while not losing any privacy.

All of the developments have an associated prototype that will be released under an open source license. As a result of the move from IMDEA to EPFL of one of the partners, the implementations of the anonymous communication modules were delayed. They require more time before they can be released.

References

[Abb+07] Timothy G. Abbott et al. "Browser-Based Attacks on Tor". In: *PETS*. 2007.

[ADN06] Jesús F. Almansa, Ivan Damgård, and Jesper Buus Nielsen. "Simplified Threshold-RSA with Adaptive and Proactive Security". In: *EUROCRYPT*. 2006.

[AG] D. F. Aranha and C. P. L. Gouvêa. *RELIC is an Efficient Library for Cryptography*. `https://github.com/relic-toolkit/relic`.

[AH16] Erinn Atwater and Urs Hengartner. "Shatter: Using Threshold Cryptography to Protect Single Users with Multiple Devices". In: *WISEC*. 2016.

[And17] Android security website. *Developing third party applications with Trusty TEE*. `https://source.android.com/security/trusty/#third-party_trusty_applications`. 2017.

[ASM06] Man Ho Au, Willy Susilo, and Yi Mu. "Constant-Size Dynamic k-TAA". In: *SCN*. 2006.

[ATK11] Man Ho Au, Patrick P. Tsang, and Apu Kapadia. "PEREA: Practical TTP-free Revocation of Repeatedly Misbehaving Anonymous Users". In: *TISSEC* (2011).

[BCC88] Gilles Brassard, David Chaum, and Claude Crépeau. "Minimum Disclosure Proofs of Knowledge". In: *J. Comput. Syst. Sci.* (1988).

[BCF17] Manuel Barbosa, Dario Catalano, and Dario Fiore. "Labeled Homomorphic Encryption: Scalable and Privacy-Preserving Processing of Outsourced Data". In: *ESORICS*. 2017.

[BDT04] Dan Boneh, Xuhua Ding, and Gene Tsudik. "Fine-grained Control of Security Capabilities". In: *TOIT* (2004).

[BG97] Mihir Bellare and Shafi Goldwasser. "Verifiable Partial Key Escrow". In: *CCS*. 1997.

[Bon+01] Dan Boneh et al. “A Method for Fast Revocation of Public Key Certificates and Security Capabilities”. In: *USENIX*. 2001.

[Boy89] Colin Boyd. “Digital Multisignatures”. In: *Cryptography and Coding* (1989).

[Bra+15] Luís T. A. N. Brandão et al. “Toward Mending Two Nation-Scale Brokered Identification Systems”. In: *PoPETs* (2015).

[Bra00] Stefan A Brands. *Rethinking public key infrastructures and digital certificates: building in privacy*. MIT Press, 2000.

[BTD12] Ero Balsa, Carmela Troncoso, and Claudia Díaz. “OB-PWS: Obfuscation-Based Private Web Search”. In: *S&P*. 2012. ISBN: 978-0-7695-4681-0.

[Bul+17] Ahto Buldas et al. “Server-Supported RSA Signatures for Mobile Devices”. In: *ESORICS*. 2017.

[Cam+06] Jan Camenisch et al. “How to Win the Clone Wars: Efficient Periodic *n*-times Anonymous Authentication”. In: *CCS*. 2006.

[Cam+16] Jan Camenisch et al. “Virtual Smart Cards: How to Sign with a Password and a Server”. In: *SCN*. 2016.

[CFN] David Chaum, Amos Fiat, and Moni Naor. “Untraceable Electronic Cash”. In: *CRYPTO ’88*.

[CG09] Richard Chow and Philippe Golle. “Faking Contextual Data for Fun, Profit, and Privacy”. In: *WPES*. 2009. ISBN: 978-1-60558-783-7.

[CH02] Jan Camenisch and Els Van Herreweghen. “Design and Implementation of the *Idemix* Anonymous Credential System”. In: *CCS*. 2002.

[CHL05] Jan Camenisch, Susan Hohenberger, and Anna Lysyanskaya. “Compact E-Cash”. In: *EUROCRYPT*. 2005.

[Cho+95] Benny Chor et al. “Private Information Retrieval”. In: *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995*. IEEE Computer Society, 1995, pp. 41–50. DOI: 10.1109/SFCS.1995.492461. URL: https://doi.org/10.1109/SFCS.1995.492461.

[CKN03] Ran Canetti, Hugo Krawczyk, and Jesper Buus Nielsen. “Relaxing Chosen-Ciphertext Security”. In: *CRYPTO*. 2003.

[CL02] Jan Camenisch and Anna Lysyanskaya. “A Signature Scheme with Efficient Protocols”. In: *SCN*. 2002.

[CLN15] Jan Camenisch, Anja Lehmann, and Gregory Neven. “Optimal Distributed Password Verification”. In: *CCS*. 2015. DOI: 10.1145/2810103.2813722.

[CW11] Scott A. Crosby and Dan S. Wallach. “Authenticated Dictionaries: Real-World Costs and Trade-Offs”. In: *ACM Trans. Inf. Syst. Secur.* 14.2 (2011), 17:1–17:30. DOI: 10.1145/2019599.2019602. URL: http://doi.acm.org/10.1145/2019599.2019602.

[Dan] George Danezis. *Petlib: a Python library that implements a number of Privacy Enhancing Technologies*. https://github.com/gdanezis/petlib. Last accessed: August 29, 2018.

[Das+18] D. Das et al. “Anonymity Trilemma: Strong Anonymity, Low Bandwidth Overhead, Low Latency — Choose Two”. In: *S&P 2018*. IEEE Computer Society, 2018, pp. 170–188. DOI: 10.1109/SP.2018.00011. URL: doi.ieeecomputersociety.org/10.1109/SP.2018.00011.

[DDC18] Sanchari Das, Andrew Dingman, and L Jean Camp. “Why Johnny Doesn’t Use Two Factor A Two-Phase Usability Study of the FIDO U2FSecurity Key”. In: *FC*. 2018.

[DDM03] George Danezis, Roger Dingledine, and Nick Mathewson. "Mixminion: Design of a Type III Anonymous Remailer Protocol". In: *S&P 2003*. 2003, pp. 2–15. ISBN: 0-7695-1940-7. URL: http://dl.acm.org/citation.cfm?id=829515.830555.

[Des87] Yvo Desmedt. "Society and Group Oriented Cryptography: A New Concept". In: *CRYPTO*. 1987.

[DF91] Yvo Desmedt and Yair Frankel. "Shared Generation of Authenticators and Signatures (Extended Abstract)". In: *CRYPTO*. 1991.

[DMS04a] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. "Tor: The Second-Generation Onion Router". In: *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*. Ed. by Matt Blaze. USENIX, 2004, pp. 303–320.

[DMS04b] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. "Tor: The Second-Generation Onion Router". In: *USENIX*. 2004.

[Duk14] V. Dukhovni. *Opportunistic Security: Some Protection Most of the Time*. RFC 7435. Dec. 2014. URL: https://rfc-editor.org/rfc/rfc7435.txt.

[EKA14] Jan-Erik Ekberg, Kari Kostiainen, and N. Asokan. "The Untapped Potential of Trusted Execution Environments on Mobile Devices". In: *S&P*. 2014.

[ElG84] Taher ElGamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". In: *CRYPTO*. 1984.

[ElG85] Taher ElGamal. "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms". In: *Trans. Inf. Theory* 31.4 (1985).

[Eve+15] Adam Everspaugh et al. "The Pythia PRF Service". In: *USENIX*. 2015.

[FS86] Amos Fiat and Adi Shamir. "How to Prove Yourself: Practical Solutions to Identification and Signature Problems". In: *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*. Ed. by Andrew M. Odlyzko. Vol. 263. Lecture Notes in Computer Science. Springer, 1986, pp. 186–194. DOI: 10.1007/3-540-47721-7_12. URL: https://doi.org/10.1007/3-540-47721-7%5C_12.

[FZ13] Matthew K. Franklin and Haibin Zhang. "Unique Ring Signatures: A Practical Construction". In: *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*. Ed. by Ahmad-Reza Sadeghi. Vol. 7859. Lecture Notes in Computer Science. Springer, 2013, pp. 162–170. ISBN: 978-3-642-39883-4. DOI: 10.1007/978-3-642-39884-1_13. URL: https://doi.org/10.1007/978-3-642-39884-1%5C_13.

[Gen+00] Rosario Gennaro et al. "Robust and Efficient Sharing of RSA Functions". In: *J. of Cryptology* (2000).

[Gen+07] Rosario Gennaro et al. "Secure Distributed Key Generation for Discrete-Log Based Cryptosystems". In: *J. Cryptology* (2007).

[GGN16] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. "Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security". In: *ACNS*. 2016.

[GLR17] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. "Message Franking via Committing Authenticated Encryption". In: *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10403. Lecture Notes in Computer Science. Springer, 2017, pp. 66–97. ISBN: 978-3-319-63696-2. DOI:

10.1007/978-3-319-63697-9_3. URL: https://doi.org/10.1007/978-3-319-63697-9%5C_3.

[Gol+18] Steven Goldfeder et al. "When the cookie meets the blockchain: Privacy risks of web payments via cryptocurrencies". In: *PoPETs* (2018).

[Haz+12] Carmit Hazay et al. "Efficient RSA Key Generation and Threshold Paillier in the Two-Party Setting". In: *CT-RSA*. 2012.

[Her15] Alex Hern. "Stagefright: new Android vulnerability dubbed 'heartbleed for mobile'". In: *The Guardian* (2015). URL: https://www.theguardian.com/technology/2015/jul/28/stagefright-android-vulnerability-heartbleed-mobile.

[Jaw+18] Husam Al Jawaheri et al. "When A Small Leak Sinks A Great Ship: Deanonymizing Tor Hidden Service Users Through Bitcoin Transactions Analysis". In: *CoRR* abs/1801.07501 (2018). arXiv: 1801.07501.

[JL13] Marc Joye and Benoît Libert. "Efficient Cryptosystems from 2^k-th Power Residue Symbols". In: *EUROCRYPT*. 2013.

[JLO97] Ari Juels, Michael Luby, and Rafail Ostrovsky. "Security of Blind Digital Signatures (Extended Abstract)". In: *CRYPTO*. 1997.

[Kim16] Kim Zetter, WIRED magazine. *How the top 5 PC makers open your laptop to hackers*. https://www.wired.com/2016/05/2036876/. 2016.

[Klu+16] Thomas Kluyver et al. "Jupyter Notebooks - a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, June 7-9, 2016*. Ed. by Fernando Loizides and Birgit Schmidt. IOS Press, 2016, pp. 87–90. DOI: 10.3233/978-1-61499-649-1-87. URL: https://doi.org/10.3233/978-1-61499-649-1-87.

[KMR12] Marcel Keller, Gert Læssøe Mikkelsen, and Andy Rupp. "Efficient Threshold Zero-Knowledge with Applications to User-Centric Protocols". In: *ICITS*. 2012.

[Kul+18] Bogdan Kulynych et al. "ClaimChain: Improving the Security and Privacy of In-band Key Distribution for Messaging". In: *WPES 2018*. 2018, pp. 86–103. DOI: 10.1145/3267323.3268947.

[Lai+16] Junzuo Lai et al. "CCA-Secure Keyed-Fully Homomorphic Encryption". In: *PKC*. 2016.

[Lip+18] Moritz Lipp et al. "Meltdown". In: *ArXiv e-prints* (Jan. 2018). arXiv: 1801.01207.

[LQ03] Benoît Libert and Jean-Jacques Quisquater. "Efficient Revocation and Threshold Pairing-based Cryptosystems". In: *PODC*. 2003.

[Mar+13] Claudio Marforio et al. "Secure Enrollment and Practical Migration for Mobile Trusted Execution Environments". In: *SPSM*. 2013.

[McG+15] Brian McGillion et al. "Open-TEE - An Open Virtual Trusted Execution Environment". In: *TrustCom*. 2015.

[Mel+15] Marcela S. Melara et al. "CONIKS: Bringing Key Transparency to End Users". In: *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. Ed. by Jaeyeon Jung and Thorsten Holz. USENIX Association, 2015, pp. 383–398. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara.

[Mie+13] Ian Miers et al. "Zerocoin: Anonymous Distributed E-Cash from Bitcoin". In: *S&P*. 2013.

[Mit05] Chris J. Mitchell. "What is trusted computing?" In: *Trusted Computing*. Vol. 6. 2005.

[MR01] Philip D. MacKenzie and Michael K. Reiter. "Networked Cryptographic Devices Resilient to Capture". In: *S&P*. 2001.

[MR04] Philip D. MacKenzie and Michael K. Reiter. "Two-party Generation of DSA Signatures". In: *Int. J. Inf. Sec.* (2004).

[MRV99] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. "Verifiable Random Functions". In: *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*. IEEE Computer Society, 1999, pp. 120–130. ISBN: 0-7695-0409-4. DOI: 10.1109/SFFCS.1999.814584. URL: https://doi.org/10.1109/SFFCS.1999.814584.

[Mul+] U. Muller et al. *Mixmaster Protocol – Version 2*. https://gnunet.org/mixmaster-spec.

[New96] Ron Newman. *The Church of Scientology vs. anon.penet.fi*. Sept. 1996. URL: https://www.spaink.net/cos/rnewman/anon/penet.html.

[ØS06] Lasse Øverlier and Paul F. Syverson. "Locating Hidden Servers". In: *S&P*. 2006.

[Pai99] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *EUROCRYPT*. 1999.

[Pap+17] Dimitrios Papadopoulos et al. "Can NSEC5 be practical for DNSSEC deployments?" In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 99. URL: http://eprint.iacr.org/2017/099.

[Ped91] Torben P. Pedersen. "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing". In: *CRYPTO*. 1991.

[Pio+17] Ania M. Piotrowska et al. "The Loopix Anonymity System". In: *USENIX*. 2017.

[PNP08] Roel Peeters, Svetla Nikova, and Bart Preneel. "Practical RSA threshold decryption for things that think". In: *WISSec 2008*. 2008.

[PR08] Manoj Prabhakaran and Mike Rosulek. "Homomorphic Encryption with CCA Security". In: *ICALP*. 2008.

[Rab98] Tal Rabin. "A Simplified Approach to Threshold and Proactive RSA". In: *CRYPTO*. 1998.

[Sch91] Claus-Peter Schnorr. "Efficient Signature Generation by Smart Cards". In: *J. Cryptology* 4.3 (1991), pp. 161–174. DOI: 10.1007/BF00196725. URL: https://doi.org/10.1007/BF00196725.

[SG02] Victor Shoup and Rosario Gennaro. "Securing Threshold Cryptosystems against Chosen Ciphertext Attack". In: *J. Cryptology* (2002).

[Sho00] Victor Shoup. "Practical Threshold Signatures". In: *EUROCRYPT*. 2000.

[SU17] Dominique Schröder and Dominique Unruh. "Security of Blind Signatures Revisited". In: *J. of Cryptol.* (2017).

[SZ05] Ravi S. Sandhu and Xinwen Zhang. "Peer-to-peer access control architecture using trusted computing technology". In: *SACMAT*. 2005.

[Tan11] O. Tange. "GNU Parallel - The Command-Line Power Tool". In: *The USENIX Magazine* 36.1 (Feb. 2011), pp. 42–47. URL: http://www.gnu.org/s/parallel.

[TDG16] Raphael R. Toledo, George Danezis, and Ian Goldberg. "Lower-Cost ϵ-Private Information Retrieval". In: *PoPETs* 2016.4 (2016), pp. 184–201. DOI: 10.1515/popets-2016-0035. URL: https://doi.org/10.1515/popets-2016-0035.

[Tsa+10] Patrick P. Tsang et al. "BLAC: Revoking Repeatedly Misbehaving Anonymous Users without Relying on TTPs". In: *TISSEC* (2010).

[Vee+16] Victor van der Veen et al. "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms". In: *CCS*. 2016.

[WRK17] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. "Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation". In: *SIGSAC*. 2017.

[Yao86] Andrew Chi-Chih Yao. "How to Generate and Exchange Secrets (Extended Abstract)". In: *FOCS*. 1986.

A Claimchain security and privacy

In this appendix we first describe how we build Unique-resolution key-value Merkle trees. Then, we formally model the security and privacy properties of the ClaimChain data structure and prove that ClaimChain satisfies these properties.

A.1 Unique-resolution key-value Merkle tree

Our unique-resolution key-value Merkle tree data structure is composed of two types of nodes:

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

We denote the root of a tree as MTR. Each Internal node contains a pivot string and the hashes of its two children. The invariant of the structure is that any nodes in the left sub-tree will have pivots or leaf keys smaller than the parent pivot, and any nodes to the right sub-tree have pivots or leaf keys equal or larger than the parent pivot. As in a normal Merkle tree, 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, or authentication proof, 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. We show that such path is indeed a proof of inclusion, and, moreover, is unique in Section A.1.2.

A.1.1 Algorithms

Building the tree To build a tree from a set of key-value pairs $S = \{..., (k_i, v_i), ...\}$ we run the BuildTree procedure (Algorithm 1) The procedure take as input a set of claims S and a content-addressable store. It constructs the tree nodes and saves them to the store. Finally, it returns the hash of the root node of the resulting tree.

Querying the tree The tree querying procedure QueryTree is described in Algorithm 2. It takes as input the tree root MTR and store that contains the tree nodes. The procedure traverses the tree starting from the root. For each intermediate node, the procedures follows a left or right sub-tree depending on the pivot field. It continues until it ends up in a leaf node. If the leaf node has the correct key, QueryTree returns the corresponding value, otherwise it returns $\perp$.

Algorithm 1 Tree construction

```
procedure BuildTree(S, store)
    if |S| = 1 then
        {(k, v)} ← S
        leaf ← Leaf(k, H(v))
        Put(store, leaf)
        Put(store, v)                          ▷ Put the value itself into the store
        return H(leaf)
    else
        (k*, v*) ←$ S                          ▷ Pick the pivot arbitrarily
        (S⁻, S⁺) ← Partition(k*, S)
        left ← BuildTree(S⁻, store)
        right ← BuildTree(S⁺, store)
        node ← Internal(k*, left, right)
        store.Put(node)
        return H(node)

procedure Partition(k*, S)
    S⁻, S⁺ ← { }, { }
    for (k, v) in S do
        if k < k* then                         ▷ Lexicographic comparison of strings
            S⁻ ← S⁻ ∪ {(k, v)}
        else
            S⁺ ← S⁺ ∪ {(k, v)}
    return (S⁻, S⁺)
```

A.1.2 Unique resolution

For a given key, only one value can be stored in the tree. Any violation of this invariant will be detected when the tree is queried—thus the creator of the tree does not need to be trusted to enforce this invariant. More formally, for a given key k it is only possible to successfully prove the inclusion of one leaf node in the tree with root MTR. We capture this notion in the UniqRes game in Experiment 1. The following theorem states that no adversary can win this game.

Theorem 3. For any probabilistic polynomial time adversary $\mathcal{A}$ it holds that $\Pr_{\mathcal{A}}[b = 1] = \mathsf{negl}(\lambda)$, where the bit $b \in \{0, 1\}$ is the output of the UniqRes game (Experiment 1).

Proof. Assume $\mathcal{A}$ wins the game. Then it is able to construct two stores such that there are two different valid paths:

$$\pi \leftarrow \textsc{GetPath}(\mathsf{MTR}, k, \mathsf{store})$$
$$\pi' \leftarrow \textsc{GetPath}(\mathsf{MTR}, k, \mathsf{store}'),$$

that start with the same root MTR, but end with different leaves containing (k, v) and (k, v') respectively.

First, assume one of the paths, w.l.o.g. π, consists of a single leaf node t with (k, v). Then the other path π' can contain either another leaf t' with (k', v), or start with an internal node t'. This implies a hash collision, since $t \neq t'$, but $\mathsf{MTR} = H(t) = H(t')$. By the collision resistance property of the cryptographic hash function H, this happens with negligible probability.

Algorithm 2 Querying the tree

```
procedure QueryTree(MTR, k, store)
    π ← GetPath(MTR, k, store)
    [..., Leaf(k′, v)] ← π
    if k′ = k then
        return ⊥
    else
        return v

procedure GetPath(h, k, store)
    node ← store.Get(h)
    if node is Leaf then
        return [node]
    else if node is Internal(pivot, left, right) then
        if k < pivot then
            π ← GetPath(left, k, store)
        else
            π ← GetPath(right, k, store)
        return [node] + π          ▷ Prepend the current node to the list π
```

Experiment 1 Unique Resolution

```
UniqRes^A(λ)
MTR, k, store, store′ ← A()
if store = store′ then
    return 0
v ← QueryTree(MTR, k, store)
v′ ← QueryTree(MTR, k, store′)
b ← v ≠ v′
return b.
```

Now, assume that the paths have a common beginning. Let t, t' be the first nodes along the paths that differ, and let $t^* = \mathsf{Internal}(p^*, h_l^*, h_r^*)$ be their common parent. Then, there are four possible options:

(a) Both t and t' are a left child of t^*. In this case, $H(t) = H(t') = h_l^*$. This implies a hash collision, which we assume to happen with negligible probability.

(b) Both t and t' are a right child of t^*. This is analogous to the previous case.

(c) The children t and t' are respectively the left child and the right child of t^*. This situation cannot happen, because GetPath decides which child to follow based on the value of the pivot p^* and the lookup key k. Since the parent is common, the procedure will always choose either the left, or the right child.

(d) The children t and t' are respectively the right child and the left child of t^*. This is analogous to the previous case.

Thus, the probability that $\mathcal{A}$ wins the game, $\Pr_{\mathcal{A}}[b = 1]$, equals the probability of a hash collision and is therefore negligible. □

Algorithm 3 Add user and extend chain oracles

▷ Add a new user
procedure AU(id)
  $(\mathsf{sk}^{\mathsf{id}}_{\mathsf{sig}}, \mathsf{pk}^{\mathsf{id}}_{\mathsf{sig}}) \leftarrow \mathsf{Sig.KeyGen}(1^\lambda)$
  $(\mathsf{sk}^{\mathsf{id}}_{\mathsf{DH}}, \mathsf{pk}^{\mathsf{id}}_{\mathsf{DH}}) \leftarrow \mathsf{DH.KeyGen}(1^\lambda)$
  $(\mathsf{sk}^{\mathsf{id}}_{\mathsf{VRF}}, \mathsf{pk}^{\mathsf{id}}_{\mathsf{VRF}}) \leftarrow \mathsf{VRF.KeyGen}(1^\lambda)$
  $\mathsf{keys}^{\mathsf{id}} \leftarrow (\mathsf{sk}^{\mathsf{id}}_{*}, \mathsf{pk}^{\mathsf{id}}_{*})$
  $(\mathsf{sk}'^{\mathsf{id}}_{\mathsf{sig}}, \ldots) \leftarrow \mathsf{keys}^{\mathsf{id}}$ ▷ Separately record the signing key

▷ Extend the chain of an existing user
procedure EC(id, data, claims, acs, store)
  if user id does not exist **then return** $\perp$
  $\mathsf{ptr}^{\mathsf{id}} \leftarrow$ ExtendChain(
      data, claims, acs, $\mathsf{keys}^{\mathsf{id}} \cup \mathsf{sk}'^{\mathsf{id}}_{\mathsf{sig}}$, $\mathsf{ptr}^{\mathsf{id}}$, store)
  return $\mathsf{ptr}^{\mathsf{id}}$

A.2 Security of the ClaimChain data structure

A.2.1 Privacy

Here we formally describe the privacy properties of ClaimChains.

Claim privacy. The adversary cannot learn anything about the content claims for which it does not have the corresponding capabilities.

We formalize this in Experiment 2 using an indistinguishability game. The game models that the adversary cannot distinguish between a claim containing one of two equal-length messages of its choice. The experiment is executed by a challenger that plays a game with the adversary $\mathcal{A}$.

The game starts with creating a user that represents an honest reader, and another user that represents the challenger. We then provide the adversary with an oracle access that allows it to create users and request them to extend their chains with adversary-supplied claims and access control sets (see Algorithm 3). Moreover, the adversary is allowed to modify store.

Eventually, the adversary outputs two claims (l_0, m_0) and (l_1, m_1). The challenger flips a random coin b, and constructs a challenge block containing claim (l_b, m_b), readable by the honest reader, but not by the adversary. The adversary then has to guess which of the two challenge claims were included in the challenge block. It may make further oracle queries.

Note that this definition implies that the adversary cannot learn anything about the claim neither from the claim encoding itself, not from any of the capabilities. Additionally, the adversary could have access to the claim in the past, but not in the challenge block.

The proof of knowledge π in the claim encoding c depends on the claim key k and other public values, making it difficult to prove directly that the adversary cannot learn anything about the bit b. Therefore, in one of the steps we replace this proof π with a completely random proof. The following lemma states that we may do so.
Lemma 3. To any distinguisher that does not know the value $k_\pi \in \{0,1\}^{2\lambda}$, the proof π in EncClaim is indistinguishable from a randomly drawn proof in the ran-

dom oracle model for H_q.

Proof. Without loss of generality, we focus on a simpler proof with only a single conjunct, writing m for $l \parallel \text{nonce}$:

$$\pi \leftarrow \mathsf{SPK}\{(\mathsf{sk}_{\mathsf{VRF}}) : \mathsf{pk}_{\mathsf{VRF}} = g^{\mathsf{sk}_{\mathsf{VRF}}} \wedge h = \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, m)\}(k_\pi).$$

Which abbreviates the following proof:

$$\pi \leftarrow \mathsf{SPK}\{(\mathsf{sk}_{\mathsf{VRF}}) : \mathsf{pk}_{\mathsf{VRF}} = g^{\mathsf{sk}_{\mathsf{VRF}}} \wedge h = H_{\mathbb{G}}(m)^{\mathsf{sk}_{\mathsf{VRF}}}\}(k_\pi).$$

To construct this proof, pick a randomizer $r_{\mathsf{sk}} \leftarrow \mathbb{Z}_p$, and compute

$$\begin{aligned} R_{\mathsf{pk}} &= g^{r_{\mathsf{sk}}} \\ R_h &= H_{\mathbb{G}}(m)^{r_{\mathsf{sk}}} \\ c &= H_p(g \parallel H_{\mathbb{G}}(m) \parallel \mathsf{pk}_{\mathsf{VRF}} \parallel h \parallel R_{\mathsf{pk}} \parallel R_h \parallel k_\pi) \\ s_{\mathsf{sk}} &= r_{\mathsf{sk}} + c \cdot \mathsf{sk}_{\mathsf{VRF}}. \end{aligned}$$

The proof is then given by (c, s_{sk}). To verify the proof, compute

$$\begin{aligned} R'_{\mathsf{pk}} &= g^{s_{\mathsf{sk}}} \mathsf{pk}_{\mathsf{VRF}}^{-c} \\ R'_h &= H_{\mathbb{G}}(m)^{s_{\mathsf{sk}}} h^{-c}, \end{aligned}$$

and verify that c equals $H_q(g \parallel H_{\mathbb{G}}(m) \parallel \mathsf{pk}_{\mathsf{VRF}} \parallel h \parallel R'_{\mathsf{pk}} \parallel R'_h \parallel k_\pi)$.

Suppose that the adversary does not know k_π. To randomly generate the proof, draw $(c', s'_{\mathsf{sk}}) \leftarrow \$\ \mathbb{Z}_p^2$ at random. Since the adversary does not know k_π it can never query the random oracle H_q with the correct value for k_π, therefore it cannot distinguish the fake proof (c', s'_{sk}) from a real proof (c, s_{sk}). □

Theorem 4 (Claim privacy)**.** For any probabilistic polynomial time adversary $\mathcal{A}$ it holds that $Pr[b = 1] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$, where $b \in \{0, 1\}$ is the result of ClaimPriv game (Experiment 2) run with $\mathcal{A}$.

Proof. We construct a sequence of games and show that $\mathcal{A}$ can distinguish between them with negligible probability, starting with $G_0 = \mathsf{ClaimPriv}^{\mathcal{A}}(\lambda)$.

First, we show that the adversary cannot extract any information about b from the capability entry for l_b because of security of the Diffie-Hellman key exchange and the encryption scheme.

Recall from the EncCap (Figure 9) and ExtendChain procedures (Figure 10) that the corresponding capability lookup key i_{cap} and the encryption key k_{cap} are given by:

$$\begin{aligned} i_{\mathsf{cap}} &= H_3(s \parallel l_b \parallel \text{nonce}) \\ k_{\mathsf{cap}} &= H_4(s \parallel l_b \parallel \text{nonce}), \end{aligned}$$

where s is the shared DH secret.

G_1 In this game we substitute the shared Diffie-Hellman secret s with the random string $\alpha \leftarrow\$\ \{0, 1\}^\lambda$ in all capabilities for reader $\mathsf{pk}_{\mathsf{DH}}^{\text{`reader'}}$ in all blocks on the challenger's chain. In particular, we set:

$$\begin{aligned} i_{\mathsf{cap}} &= H_3(\alpha \parallel l_b \parallel \text{nonce}) \\ k_{\mathsf{cap}} &= H_4(\alpha \parallel l_b \parallel \text{nonce}), \end{aligned}$$

Experiment 2 Claim privacy

$\underline{\mathsf{ClaimPriv}^{\mathcal{A}}(\lambda)}$

Setup

AU('reader') ▷ Initialize reader's chain

AU('challenger') ▷ Initialize challenger's chain

Content to include in the challenge block

$(l_0, m_0), (l_1, m_1), \mathsf{data}, \mathsf{claims}), \mathsf{acs}, \mathsf{store}) \leftarrow$

$\leftarrow \mathcal{A}^{\mathsf{EC}(\cdot),\mathsf{AU}(\cdot)}\left(\mathsf{pk}_{\mathsf{DH}}^{\text{'reader'}}\right)$

if l_0 or l_1 in acs or $|m_0| \neq |m_1|$ **then return** 0

Challenge block

$b \leftarrow\$\ \{0,1\}$

$\mathsf{claims}' \leftarrow \mathsf{claims} \cup \{(l_b, m_b)\}$

$\mathsf{acs}' \leftarrow \mathsf{acs} \cup \{(\mathsf{pk}_{\mathsf{DH}}^{\text{'reader'}}, l_b)\}$ ▷ Give the reader the access to l_b

$\mathsf{ptr}_C \leftarrow \mathsf{EC}(\text{'challenger'}, \mathsf{data}, \mathsf{claims}', \mathsf{acs}', \mathsf{store})$

Response

$\hat{b} \leftarrow \mathcal{A}^{\mathsf{EC}(\cdot),\mathsf{AU}(\cdot)}\left(\mathsf{ptr}_C\right)$

return $\hat{b} = b$

G_2 In this game, we substitute the capability key k_{cap} with a random string $\beta \leftarrow\$\ \{0,1\}^{2\lambda}$. The capability becomes:

$$\mathsf{cap} = E_(\beta, h \parallel k \parallel k_\pi).$$

G_3 In this game, we substitute the lookup index i_{cap} with a random string $\gamma \leftarrow\$\ \{0,1\}^{2\lambda}$ as well.

G_4 In this game, we substitute the plaintext $h \parallel k \parallel k_\pi$ with a random string γ of the same length:

$$\mathsf{cap} = E_(\beta, \gamma).$$

The games G_0 and G_1 are indistinguishable by the decisional Diffie-Hellman assumption. Games G_1 and G_2 are indistinguishable by the pseudorandomness of the hash function H_4. The indistinguishability of G_2 and G_3 follows from the pseudorandomness of H_3. Since the encryption key β is random, distinguishing between G_3 and G_4 can be trivially reduced the IND-CPA security for the encryption scheme. Therefore, games G_3 and G_4 are indistinguishable as well.

The adversary is not allowed to give access to labels l_0, l_1 to any user (honest or not). As a result, no other capability entries depend on the challenge bit b.

Since G_4 replaces the real plaintext with a random plaintext, the adversary also does not learn anything about k and k_π.

Now we show that the adversary cannot extract information about b neither from the claim encoding, nor from the claim lookup key. We use the IND-CPA security of the encryption scheme and pseudorandomness of the VRF scheme.

Recall from the EncClaim (Figure 9) and ExtendChain procedures (Figure 10) that here the encoded claim c is given by:

$$c = E_(k, \pi \parallel m_b) \parallel \mathsf{com}.$$

G_5 In this game, we replace the non-interactive zero-knowledge proof π with a uniformly random proof π' that does not depend on any of the secret values, nor on any of the public values.

G_6 In this game, we replace the commitment com by a random commitment $\mathsf{com}_R \leftarrow \mathbb{G}$.

Games G_4 and G_5 are indistinguishable because of Lemma 3. Since π_{com} no longer depends on the randomness r, the commitment com is perfectly hiding. Therefore, games G_5 and G_6 are indistinguishable as well.

Next, we change the claim encryption key k to a random key. Note that because of the changes made in G_4, the adversary does not learn anything about k from the capability cap.

G_7 In this game, we generate a random encryption key δ and use it to replace k:

$$c = E_(\delta, \pi' \parallel m_b) \parallel \mathsf{com}.$$

G_8 In this game, we replace the plaintext $\pi' \parallel m_b$ with a random message μ of the same length:

$$c = E_(\delta, \mu) \parallel \mathsf{com}_R.$$

Games G_6 and G_7 are indistinguishable since the adversary learns nothing about k because of earlier transformations. Games G_7 and G_8 are indistinguishable because of the CPA security of the encryption scheme.

The final dependency on the bit b is in the claim lookup key $i = H_1(h_b)$, see EncClaim (Figure 9). We remove this final reference.

G_9 In this game, we substitute h_b in i with a random value $q' \leftarrow\$ \mathbb{G}$:

$$i = H_1(q')$$

The changes in games G_4 and G_5 ensure that the adversary does not learn anything about h_b directly. Also, indirectly the adversary cannot learn about h_b. The adversary can learn *other* VRF values by adding claims and giving itself access to them. However, the pseudorandomness property of the VRF ensures that even if the adversary makes many VRF queries, the remaining values remain pseudorandom. Hence, the adversary cannot distinguish G_8 from G_9.

In game G_9 none of the values depend on the challenge bit b, hence, the adversary cannot have advantage better than random guessing. □

Capability-reader unlinkability. The adversary should not be able to determine who has been given access to a claim, i.e., for which honest user a capability has been created. We model this using the indistinguishability game in Experiment 3. The adversary can create users (using the AU oracle) and extend their chains (using the EC oracle). It then outputs the public keys $\mathsf{pk}^0_{\mathsf{DH}}$ and $\mathsf{pk}^1_{\mathsf{DH}}$ of two honest users it created using the AU and a description of a claim with label l on which it wants to be challenged. The challenger picks one of the honest users at random, and adds a capability to l for that user. The adversary must decide which user has been given the capability.

Experiment 3 Capability-reader unlinkability

$\mathsf{CapReaderUnlink}^{\mathcal{A}}(\lambda)$

Setup

AU('challenger') ▷ Initialize challenger's chain

Content to include in the challenge block

$\mathsf{pk}^0_{\mathsf{DH}}, \mathsf{pk}^1_{\mathsf{DH}}, l, m, \mathsf{data}, \mathsf{claims}, \mathsf{acs}, \mathsf{store} \leftarrow \mathcal{A}^{\mathsf{EC}(\cdot),\mathsf{AU}(\cdot)}()$

if $\mathsf{pk}^0_{\mathsf{DH}}$ or $\mathsf{pk}^1_{\mathsf{DH}}$ not a honest user **then return** 0

Challenge block

$b \leftarrow\$ \ \{0,1\}$

$\mathsf{claims}' \leftarrow \mathsf{claims} \cup \{(l, m)\}$

$\mathsf{acs}' \leftarrow \mathsf{acs} \cup \{(\mathsf{pk}^b_{\mathsf{DH}}, l)\}$

$\mathsf{ptr}_C \leftarrow \mathsf{EC}(\text{'challenger'}, \mathsf{claims}', \mathsf{acs}', \mathsf{store})$

Response

$\hat{b} \leftarrow \mathcal{A}^{\mathsf{EC}(\cdot),\mathsf{AU}(\cdot)}(\mathsf{ptr}_C)$

return $\hat{b} = b$

Theorem 5. For any polynomially-bounded $\mathcal{A}$ it holds that $Pr[b = 1] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$, where $b \in \{0, 1\}$ is the result of CapReaderUnlink game (Experiment 3).

Proof. We show that the adversary cannot extract any information about b from the capability entry for l. The adversary may have given other readers access to label l, but the corresponding capabilities are independent of the bit b, so we ignore them. We focus instead on the capability for reader $\mathsf{pk}^b_{\mathsf{DH}}$. Recall from the EncCap (Figure 9) and ExtendChain procedures (Figure 10) that the corresponding capability lookup key i_{cap} and the encryption key k_{cap} are given by:

$$i_{\mathsf{cap}} = H_3(s \parallel l_b \parallel \mathsf{nonce})$$
$$k_{\mathsf{cap}} = H_4(s \parallel l_b \parallel \mathsf{nonce}),$$

where s is the DH secret between the chain owner and the reader $\mathsf{pk}^b_{\mathsf{DH}}$. We apply the sequence of games $G_0, \ldots, G_4$ in the proof of Theorem 4. The indistinguishability of the games proves that the adversary does not learn anything about the bit b. Therefore, we have capability-reader unlinkability. □

A.2.2 Non-equivocation

Intra-block non-equivocation. Within a given block, a ClaimChain owner cannot include two different claims having the same label to different readers.

We model this in Experiment 4. The adversary's task is to produce a block pointed to by ptr and a label l such that the two readers $\mathsf{pk}_{\mathsf{DH}}$ and $\mathsf{pk}'_{\mathsf{DH}}$ derive different claims m and m'.

Theorem 6 (Intra-block non-equivocation)**.** For any polynomially-bounded $\mathcal{A}$ it holds that $Pr[b = 1] \leq \mathsf{negl}(\lambda)$, where $b \in \{0, 1\}$ is the result of BlockNonEq game (Experiment 4).

Experiment 4 Intra-block non-equivocation

$\mathsf{BlockNonEq}^{\mathcal{A}}(\lambda)$

$\mathsf{sk}_{\mathsf{DH}}, \mathsf{pk}_{\mathsf{DH}} \leftarrow \mathsf{DH.KeyGen}(1^{\lambda})$
$\mathsf{sk}'_{\mathsf{DH}}, \mathsf{pk}'_{\mathsf{DH}} \leftarrow \mathsf{DH.KeyGen}(1^{\lambda})$
$l, \mathsf{ptr}, \mathsf{store}, \mathsf{store}' \leftarrow \mathcal{A}(\mathsf{pk}_{\mathsf{DH}}, \mathsf{pk}'_{\mathsf{DH}})$
$m \leftarrow \textsc{GetClaim}(\mathsf{sk}_{\mathsf{DH}}, l, \mathsf{ptr}, \mathsf{store})$
$m' \leftarrow \textsc{GetClaim}(\mathsf{sk}'_{\mathsf{DH}}, l, \mathsf{ptr}, \mathsf{store}')$
return $m \neq m' \wedge m \neq \bot \wedge m' \neq \bot$

Proof. We first prove that both store and store′ must contain the same block B. Suppose not, i.e., store contains block B whereas store′ contains a different block B' that both hash to the same head ptr. Then the adversary breaks the collision resistance of H. Since H is a cryptographic hash function, this happens with negligible probability.

The remainder of this proof is also by contradiction. Assume adversary $\mathcal{A}$ wins Experiment 4. We use the uniqueness of the VRF, first for the claim key k, then for the lookup key h, to derive a contradiction, i.e., that $m = m'$. Both readers $\mathsf{pk}_{\mathsf{DH}}$ and $\mathsf{pk}'_{\mathsf{DH}}$ first compute the capability lookup key (step 3), see GetClaim procedure (Figure 10), retrieve the capability (step 4) and decode it (step 5). Capabilities are per reader, and therefore different. We continue the proof from step 5.

Let i and i' be the claim lookup keys derived in step 5 of the GetClaim() call by respectively the first and second user. We first consider the case where $i = i'$. By the unique resolution property of the tree (see Experiment 1), we know that in step 6 both GetClaim() calls must then derive the same claim encoding c with overwhelming probability.

Since the adversary wins, the derived messages m and m' are different and not $\bot$, therefore the calls to DecClaim() in step 7 returned different messages $m \neq m'$:

$$m \leftarrow \textsc{DecClaim}(\mathsf{pk}^{O}_{\mathsf{VRF}}, l, h, k, k_{\pi}, c, \mathsf{nonce})$$
$$m' \leftarrow \textsc{DecClaim}(\mathsf{pk}^{O}_{\mathsf{VRF}}, l, h', k', k'_{\pi}, c, \mathsf{nonce}).$$

Since the encoding c is the same for both m and m', this situation is not possible by the binding property of the commitment scheme. Indeed, the users verify proofs π, respectively π', in step 6, which verify the commitment com.

We now consider the case where the readers derive different lookup keys i and i' in step 5. Since $i \neq i'$ and by the collision resistance of H_1, we have that the corresponding VRF values h and h' must be different as well. However, by uniqueness of the VRF, this cannot happen. More precisely, both users successfully verify the proofs π, respectively π', in step 6, which prove that $h = \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, l \parallel \mathsf{nonce}) = k'$, respectively $h' = \mathsf{VRF.Eval}(\mathsf{sk}_{\mathsf{VRF}}, l \parallel \mathsf{nonce})$, and therefore $h = h'$, contradicting the assumption that $i \neq i'$. □

Detectable inter-block equivocation. The game in Experiment 5 models that a claim owner cannot make a non-consistent reference, yet produce a proof of consistency that validates using CheckConsistency() (see Figure 13). More precisely, the adversary outputs valid blocks on two chains: the blocks $\{O_i\}_1^n$ on its own chain, and the blocks $\{C_i\}_1^t$ on the referenced chain. Moreover, the adversary outputs a label l for the referenced chain, and a valid consistency proof π_{consist}.

Experiment 5 Detectable inter-block equivocation

$\mathsf{InterBlockEqDetection}^{\mathcal{A}}(\lambda)$

Setup

AU('challenger') ▷ Initialize the challenger's chain

Adversary-supplied blocks and validation of consistency

$\{O_i\}_1^n, \{C_i\}_1^t, \mathsf{store}, l, \pi_{\mathsf{consist}} \leftarrow \mathcal{A}^{\mathsf{AU}(\cdot),\mathsf{EC}(\cdot)}()$

if ValidateBlocks$(\{O_i\}_1^n) = \perp$ **then return** 0

if ValidateBlocks$(\{C_i\}_1^t) = \perp$ **then return** 0

if CheckConsistency$(l, \{O_i\}_1^n, \{C_i\}_1^t, \pi_{\mathsf{consist}}) = \perp$ **then**
  return 0

Final read phase

$\mathsf{ptr}, \mathsf{store}' \leftarrow \mathcal{A}()$

if Get$(\mathsf{store}', \mathsf{ptr}) \notin \{O_i\}_1^n$ **then return** 0

$m \leftarrow$ GetClaim$(\mathsf{sk}_{\mathsf{DH}}^{\text{'challenger'}}, l, \mathsf{ptr}, \mathsf{store}')$

return $m \notin \{C_i\}_1^t$

To win, the adversary also outputs a pointer ptr to one of its own blocks such that the challenger has access to label l. The adversary wins if the cross-referenced block m differs from the legitimate cross referenced blocks $\{C_i\}_1^t$.

Theorem 7. For any polynomially-bounded stateful $\mathcal{A}$ it holds that $Pr[d = \top] = \mathsf{negl}(\lambda)$, where $d \in \{\top, \perp\}$ is the result of DetEq game (Experiment 5).

Proof. Suppose the adversary wins the game. Let i be the index such that ptr corresponds to block O_i. Since the adversary wins,

$$m = \text{GetClaim}(\mathsf{sk}_{\mathsf{DH}}^{\text{'challenger'}}, l, \mathsf{ptr}, \mathsf{store}')$$

returned a message $m \notin \{C_i\}_1^t$. Let h be the VRF hash that it computes in step 5, and let $c_i = \bar{c}_i \parallel \mathsf{com}_i$ be the encoded claim that this algorithm retrieves in step 6. In step 7, the algorithm calls DecClaim(), to verify the proof π. Since the proof is valid, com_i commits to $H_q(m)$ and h_i is the VRF hash of $l \parallel \mathsf{nonce}_i$.

We now show that CheckConsistency() retrieves the same commitment com_i together with a proof that the committed value $x' \in \{H_q(C_i)\}_1^t$, contradicting the binding property of the commitment scheme.

The proof π_{consist} contains the VRF hash h_i' of $l \parallel \mathsf{nonce}_i$ and the proof of correctness $\pi_h^{(i)}$. Since the proof verified, h_i' is the VRF hash of $l \parallel \mathsf{nonce}_i$, and therefore $h_i' = h_i$. By the unique resolution property of the tree, CheckConsistency() therefore derived the same encoded claim $c_i = \bar{c}_i \parallel \mathsf{com}_i$ as the challenger did by calling GetClaim(). Moreover, the proof $\pi_{\mathsf{ref}}^{(i)}$ proves that com_i commits to x' such that $x' \in \{H_q(C_i)\}_1^t$.

This contradicts the binding property of the commitment scheme or the soundness of the zero-knowledge proofs. □

B Tandem security and privacy

In this appendix we prove the security and privacy properties of Tandem.

B.1 Proofs of Lemmas

Proof of Lemma 1. Whenever a ciphertext c_i is selected by the TS for opening, the TS checks that it and the corresponding randomizers κ_i, δ_i, ξ_i, and r_i are as in equation (3) and that $\delta_i < 2^{\ell_\delta}$, and hence as stated in the theorem.

Since the TS checks k tuples, every adversary needs to include at least k correct tuples in its set of $2k$ tuples. If no index i^* exists for the remaining tuples, then all k of them were incorrectly formed. The probability that none of these k bad tuples were selected during the cut and choose protocol is $1/\binom{2k}{k}$. □

Proof of Lemma 2. From Lemma 1 we know that with probability $1 - 1/\binom{2k}{k}$ there exists i^* and δ^*, x_S such that

$$\mathbf{D}^+_{sk}(c_{i^*}) = x_S + \delta^*$$

Let $c_j = \mathbf{E}^+_{pk}(\alpha)$. From equation (4) we know that:

$$c_j = c_{i^*} \cdot \mathbf{E}^+_{pk}(\gamma_{i^*}; \kappa_{i^*})$$

By decrypting we find that $\alpha = x_S + \delta^* + \gamma_{i^*} \pmod N$. Moreover, $\delta^* < 2^{\ell_\delta}$ (by Lemma 1), $x_S < p < 2^{\ell_\delta}$ (by construction) and $\gamma_{i^*} < 2^{\ell_\delta}$ as checked by the TS. Since $\ell_\delta = \lceil \log p \rceil + \ell + \log k + 2$ and $N > 3 \cdot 2^{\ell_\delta}$, we have that $\alpha = x_S + \delta^* + \gamma_{i^*}$ as integers, and thus c_j is a proper randomization, with randomizer $\delta^* + \gamma_{i^*} < 2^{\ell_\delta + 1}$, of x_S as well. □

B.2 Constructing Correctness Proof of $\overline{x_S}$

In this section we describe the details of the range proof of $\mathbf{D}^+_{sk}(\overline{x_S})$ in the Register-User protocol. The range proof ensures that the TS cannot recognize anonymous users by constructing specially crafted versions of $\overline{x_S}$ as explained earlier. When using a homomorphic encryption scheme that supports zero-knowledge proofs, such as Paillier's encryption scheme, we can use standard techniques, see for example the bitwise technique by Bellare and Goldwasser [BG97], to prove that $\mathbf{D}^+_{sk}(\overline{x_S})$ is at most 2ℓ bits (which is a sufficient proxy for p in our schemes).

In our implementation, however, we use Joye and Libert's encryption scheme which does not readily allow zero-knowledge proofs. Therefore, we instantiate the range proof using a construction that consists of two parts.

I. The TS constructs a commitment C to x_S using a commitment scheme whose message space is at least as big as the plaintext space of the encryption scheme. The TS then uses a traditional zero-knowledge proof to show that the value x_S committed in C is smaller than p.

II. Next, the TS uses a cut-and-choose technique to show that C commits to $\mathbf{D}^+_{sk}(\overline{x_S}) = x_S$.

The details are as follows. The user and TS take $\overline{x_S}$ as input. The TS takes as private input x_S and the randomizer κ used to construct $\overline{x_S}$. Let $\overline{\mathbb{G}}$ be a cyclic group of order $\overline{p}$ generated by $\overline{g}$ such that $\overline{p} > N$ (recall, N is the size of the plaintext domain of the homomorphic encryption scheme). Let $\overline{h}$ be another generator of $\overline{\mathbb{G}}$ such that the discrete logarithm of $\overline{h}$ with respect to $\overline{g}$ is unknown. We use this group to create a commitment scheme with a large message space.

The details of the first step are as follows. Part I is represented by step 1, whereas part II is represented by the cut-and-choose technique in steps 2 – 7. If at any step a verification fails, the protocol is aborted. The cut-and-choose technique is very similar to the construction we use in the ObtainKeyShareToken and GenShares protocols. Let k be the difficulty level of the cut-and-choose protocol.

1. The TS creates a non-interactive proof that the commitment C contains key-share x_S of the correct size:

$$PK\{(x_S, r) : C = \overline{g}^{x_S}\overline{h}^{r} \wedge x_S \in [0, p)\}. \tag{7}$$

   and sends this proof to the user. This proof can be implemented using a standard technique like the bitwise commitment technique of Bellare and Goldwasser [BG97]. The user checks the correctness of the proof.

2. The user randomly chooses a subset $\mathcal{D} \subset \{1, \ldots, 2k\}$ of cardinality k. She commits to $\mathcal{D}$ by picking $\theta \in_R \{0,1\}^{\ell}$ and sending $\Delta = \mathsf{ExtCommit}(\mathcal{D}, \theta)$ to the TS.

3. The TS picks randomizers $\delta_1, \ldots, \delta_{2k} \in_R \{0,1\}^{\ell_\delta}$ and $\kappa_1, \ldots, \kappa_{2k} \in_R \mathcal{R}$ to construct ciphertexts, and $r, r_1, \ldots, r_{2k} \in \mathbb{Z}_{\overline{p}}$ to create commitments. Then, the TS computes a commitment $C = \overline{g}^{x_S}\overline{h}^{r}$ and sets:

$$\begin{aligned} c_i &= \mathbf{E}^{+}_{pk}(\delta_i; \kappa_i) \\ C_i &= \overline{g}^{\delta_i}\overline{h}^{r_i} \end{aligned} \tag{8}$$

   for $i = 1, \ldots, 2k$. Finally, the TS sends the ciphertexts $c_1, \ldots, c_{2k}$ and commitments $C, C_1, \ldots, C_{2k}$ to the user. The commitments are computationally binding and information theoretically hiding. (Contrary to the ObtainKeyShareToken protocol, the TS can safely send the ciphertexts, because the user cannot decrypt them.)

4. The user sends the subset $\mathcal{D}$ and the commitment randomizer θ to the TS.

5. If $\Delta = \mathsf{ExtCommit}(\mathcal{D}, \theta)$, then the TS sends $(\delta_i, \kappa_i, r_i)_{i \in \mathcal{D}}$ to the user (otherwise, it aborts). The user verifies that the values c_i, C_i for $i \in \mathcal{D}$ satisfy equation (8). Moreover, the user checks that $\delta_i < 2^{\ell_\delta}$ for $i \in \mathcal{D}$.

6. Next, the TS computes

$$\gamma_i = \delta_i - x_S, \quad \rho_i = r_i - r, \quad \nu_i = \kappa_i \kappa^{-1}$$

   for $i \notin \mathcal{D}$, and sends them to the user.

7. Finally, the user checks that

$$\begin{aligned} c_i &= \overline{x_S} \cdot \mathbf{E}^{+}_{pk}(\gamma_i; \nu_i) \\ C_i &= C \cdot \overline{g}^{\gamma_i}\overline{h}^{\rho_i} \end{aligned} \tag{9}$$

   and that $0 \leq \gamma_i < 2^{\ell_\delta}$ for $i \notin \mathcal{D}$, and accepts the proof if all verifications are correct.

Lemma 4. If the user does not reject in the above protocol, then with probability $1 - 1/\binom{2k}{k}$ we have that $\mathbf{D}^+_{sk}(\overline{x_S}) \in [0, p)$ as required.

Proof. From the zero-knowledge proof in step 1, we know that the TS knows an opening α', r' of $C = \overline{g}^{\alpha'}\overline{h}^{r'}$ such that $0 \leq \alpha' < p$. We complete the proof by showing that $\alpha' = \mathbf{D}^+_{sk}(\overline{x_S})$.

We continue as per Lemma 1 and Lemma 2. We restate them here for completeness. First, along the lines of Lemma 1, with probability $1 - 1/\binom{2k}{k}$ there exists an index i^* such that the TS knows an opening δ^*, r^* such that:

$$\begin{aligned} \delta^* &= \mathbf{D}^+_{sk}(c_{i^*}) < 2^{\ell_\delta} \\ C_{i^*} &= \overline{g}^{\delta^*}\overline{h}^{r^*}. \end{aligned} \tag{10}$$

The user checks that the TS knows an opening for the k pairs that are opened by the TS in step 4. So, the TS must include at least k pairs for which it knows a correct opening. Suppose, for contradiction, that the index i^* does not exist, i.e., that the remaining k pairs are incorrect or cannot be opened by the TS. Since the protocol completed, the user did not detect foul play. This situation can only occur if the TS correctly guesses the set $\mathcal{D}$ in advance. Since the TS does not learn anything about $\mathcal{D}$ before step 3, the probability that none of the remaining pairs is correct is $1/\binom{2k}{k}$, as required.

Assume now that this index i^* as required above exists. We use this to show that C commits to $\mathbf{D}^+_{sk}(c)$, i.e., that $\alpha' = \mathbf{D}^+_{sk}(c)$. From equation (9) we know that:

$$C_{i^*} = C \cdot \overline{g}^{\gamma_{i^*}}\overline{h}^{\rho_{i^*}}$$

so, by using equation (10) and equating exponents, we find that $\delta^* = \alpha' + \gamma_{i^*} \pmod{\overline{p}}$. We know from the zero-knowledge proof that $\alpha' < p$ and by direct inspection that $\gamma < 2^{\ell_\delta}$ therefore, the equality holds over the integers as well, and we have

$$\delta^* = \alpha' + \gamma_{i^*} < 2^{\ell_\delta + 1} < N. \tag{11}$$

From equation (9) we also know that:

$$c_{i^*} = \overline{x_S} \cdot \mathbf{E}^+_{pk}(\gamma_{i^*}; \nu_{i^*})$$

By decrypting and using equation (10) we find that:

$$\delta^* = \mathbf{D}^+_{sk}(\overline{x_S} \cdot \mathbf{E}^+_{pk}(\gamma_{i^*}; \nu_{i^*})) = \mathbf{D}^+_{sk}(\overline{x_S}) + \gamma_{i^*} \pmod{N}.$$

Substituting δ^* from equation (11) and substracting γ_{i^*} shows that $\alpha' = \mathbf{D}^+_{sk}(\overline{x_S}) \pmod{N}$, and therefore, by size of α' and $\mathbf{D}^+_{sk}(\overline{x_S}) < N$, that $\alpha' = \mathbf{D}^+_{sk}(\overline{x_S})$ as required. □

In the security proof, we replace $\overline{x_S}$ with the encryption of 0, so that the adversary who has corrupted a user learns nothing about x_S (except what is revealed as a result of the threshold-cryptographic protocol). The following lemma states that we can do so, without the adversary detecting this change.

Lemma 5. TS can simulate the correctness proof given above such that $\overline{x_S} = \mathbf{E}^+_{pk}(0)$, provided that the encryption scheme is CPA secure and the commitment scheme $\mathsf{ExtCommit}(\cdot, \cdot)$ is extractable. This simulation does not require any knowledge of how $\overline{x_S}$ was created.

This proof uses a sequence of games that interpolates between the situation where the RegisterUser protocol is executed normally, and the situation, where $\overline{x_S}$ is an encryption of 0. This game is as in the security game: the adversary can make RegisterUser, ObtainKeyShareToken, GenShares, and BlockShare queries. It's task is to determine if $\overline{x_S}$ is as in the original protocol, or $\overline{x_S} = \mathbf{E}^+_{pk}(0)$. In particular:

- Game 0. In Game 0, $\overline{x_S}$ is constructed as per the protocol.
- Game 1. We proceed as in Game 0, but simulate the cut-and-choose proof in steps 2 – 7 by extracting $\mathcal{D}$.
- Game 2. As in Game 1, but simulate the zero-knowledge proof in step 1 of the protocol.
- Game 3. As in Game 2, but replace the commitment C by a random commitment.
- Game 4. As in Game 3, but replace $\overline{x_S}$ with an encryption of 0.

We show that each pair of consecutive games is indistinguishable to a polynomially-bounded adversary. Hence, no adversary can distinguish Game 0 from Game 4, thus proving the lemma.

Proof of Lemma 5. We first show how to simulate the cut-and-choose proof in steps 2 – 7. The adverary sends a commitment Δ to the TS in step 1. We use the extractability of $\mathsf{ExtCommit}(\cdot,\cdot)$ to recover $\mathcal{D}$ from Δ (for example, using the random oracle model if it is implemented using a hash-function).

We change how TS acts in step 3. Let $\mathcal{D} \subset \{1,\ldots,2k\}$ be the subset of cardinality k extracted from Δ. For all $i \in \mathcal{D}$ the TS sets c_i and C_i as per equation (8). For other elements, i.e., for $i \in \{1,\ldots,2k\} \setminus \mathcal{D}$, the TS generates $\gamma \in_R \{0,\ldots,2^{\ell_\delta}-1\}, \rho \in_R \mathbb{Z}_{\overline{p}}, \nu \in_R \mathcal{R}$ and sets c_i and C_i as per equation (9).

In step 4, the adversary reveals $\mathcal{D}$ and θ. If $\Delta = \mathsf{ExtCommit}(\mathcal{D},\theta)$ then with overwhelming probability, we correctly extracted $\mathcal{D}$. If we correctly extracted $\mathcal{D}$, the TS can open the tuples for $i \in \mathcal{D}$ in step 5 and return γ_i, ρ_i, ν_i for the other elements. Both satisfy the adversary's checks in steps 5 and 7.

Games 0 is indistinguishable from Game 1. The simulated proof can go wrong for two reasons. One, we can fail to extract the disclose set $\mathcal{D}$, but this can only happen with negligible probability. Second, the distribution of γ_is for $i \notin \mathcal{D}$ is not completely correct, however, the size of δ ensures that this difference is statistically hidden from the adversary. So, from the point of view of the adversary, Games 0 and 1 are indistinguishable.

In Game 2 we simulate the zero-knowledge proof in step 1. By construction of the simulator of this proof, the adversary cannot detect this change.

As a result of the changes we made in Game 1, the answers of TS do not depend on the opening of C. So, in Game 3 the TS can generate a random commitment $C \in_R \overline{\mathbb{G}}$. Since Pedersen's commitment scheme is information-theoretically hiding, the adversary cannot detect this change.

In Game 4, the TS sends $\overline{x_S} = \mathbf{E}^+_{pk}(0)$ to the user instead of an encryption of the key-share x_S. As a result of the changes we made in Game 1, the TS can still complete the remaining part of the protocol.

We claim that the adversary $\mathcal{A}$ cannot distinguish Games 3 and 4. Suppose to the contrary that $\mathcal{A}$ *can* distinguish Games 3 and 4. We then show that $\mathcal{A}$ can break the CPA security of the homomorphic encryption scheme.

To do so, we build an adversary $\mathcal{B}$ against the CPA security of the encryption scheme. Recall that $\mathcal{B}$ can make a challenge query on two messages m_0 and m_1. In our case, $\mathcal{B}$ picks $m_0 = x_S$ and $m_1 = 0$. Then, its challenger returns a ciphertext $c_* = \mathbf{E}^+_{pk}(pk, m_b)$ for some bit $b \in_R \{0, 1\}$. Adversary $\mathcal{B}$ needs to guess b.

In RegisterUser queries for the challenge user U^*, adversary $\mathcal{B}$ (which acts as a challenger to $\mathcal{A}$) uses $\overline{x_S} = c_*$. Clearly, if $b = 0$, then $\mathcal{B}$ perfectly simulates Game 3. If $b = 1$, it perfectly simulates Game 4. Therefore, if $\mathcal{A}$ can distinguish between Games 3 and 4, it can break the CPA security of the encryption scheme. □

B.3 Security Proof

In the security proof, the challenger controls the TS and the adversary tries to attack a user. The security proof is a sequence of games. In the final game, the challenger simulates the game using only the TCP oracle of the TCP security game, without knowing the corresponding TS' key-share x_S. As a result, any adversary that manages to use the blocked key of that user must therefore break the security of the underlying threshold-cryptographic protocol.

We use the following sequence of games:

- Game 0: We play the game as described in the TANDEM Security game, see Game 1 on page 53.
- Game 1: We change the definition of GenShares. The challenger simulates the workings of TS but does not decrypt any ciphertext. Instead, the TS uses the extractability of ExtCommit$(\cdot, \cdot)$ and the Δ_is (from the corresponding ObtainKeyShareToken protocol) to compute the plaintext corresponding to c_j (without decrypting), which it uses as $\tilde{x}_S$. Finally, the TS constructs the proof of knowledge of $\tilde{x}_S$ as before.
- Game 2: We guess the challenge user U^* and we change the definition of RegisterUser for this user: we replace $\overline{x_S} = \mathbf{E}^+_{pk}(x_S)$ by $\overline{x_S} = \mathbf{E}^+_{pk}(0)$.
- Game 3: For all non-challenge users we answer GenShares queries as in the previous game. For U^* the TS simulates the TCP following GenShares using the TCP security oracle (without knowing x_S of U^*).

We then prove the following:

- The adversary cannot distinguish Game 0 from Game 1. We prove that as long as one of the tuples is as it should be—and Lemma 1 shows that this is the case with high probability—then we correctly recover the plaintext of c_j and thus the TS extracts the correct $\tilde{x}_S$, and therefore the TCP is correct as well.
- The adversary cannot distinguish Game 1 from Game 2. We no longer decrypt ciphertexts. Hence, we can use the CPA security of the encryption scheme to show that the adversary cannot distinguish Game 1 from Game 2. More formally, we build a distinguisher that interpolates between Games 1 and 2. The distinguisher makes a query for $m_0 = x_S$ and $m_1 = 0$ to its CPA challenger, and uses the answer as $\overline{x_S}$. Lemma 5 shows the adversary cannot detect this change to RegisterUser. If the CPA challenger returned an

encryption of x_S then the distinguisher perfectly simulates Game 1, otherwise it simulates Game 2. We can still answer GenShares queries correctly, since we no longer need to decrypt any ciphertexts.

- The adversary cannot distinguish Game 2 from Game 3 because the TCP oracle simulates the same protocol.
- Finally, if we have an adversary that can win Game 3, then it breaks the security of the TCP because by construction the challenger has no new tokens for the challenge user U^* (because she is blocked or rate-limited) in the challenge phase.

Proof of Theorem 1. This proof follows the sequence of games highlighted above. Let U^* be the challenge user. We guess this user. If the guess turns out to be incorrect, we repeat the reduction with a new guess.

In Game 1 we change how the TS responds to RunTCP queries, in particular, we change GenShares for the challenge user U^*. The TS (controlled by the challenger) no longer decrypts the ciphertext c_j revealed in a token, but instead directly recovers the plaintext using the Δ_i values. The TS then continues as before.

To enable the TS to answer RunTCP queries without decrypting, the TS stores some extra values whenever $\mathcal{A}$ runs the ObtainKeyShareToken protocol. Whenever the TS blindly signs a token, it extracts, *id*, the token's identifier (normally, the TS cannot learn this value). The challenger uses the extractability of $\mathsf{ExtCommit}(\cdot,\cdot)$ to find inputs $\delta'_{i_1},\ldots,\delta'_{i_m}$ and $\kappa'_{i_1},\ldots,\kappa'_{i_m}$ used to create the unopened commitments $\Delta_{i_1},\ldots,\Delta_{i_m}$. (The adversary might cheat so that not all Δ_is are true commitments.) By Lemma 1, $m \geq 1$, and there exists i^* such that the extracted inputs are correct, i.e., $\delta'_{i^*} = \delta_{i^*}$ and $\kappa'_{i^*} = \kappa_{i^*}$. The challenger records the tuple $(id, U, (i_1, \delta'_{i_1}, \kappa'_{i_1}), \ldots, (i_m, \delta'_{i_m}, \kappa'_{i_m}))$ for later use.

We now show how to answer RunTCP queries without needing to decrypt the ciphertexts. The TS initially follows the GenShares protocol. At the start of the protocol, $\mathcal{A}$ sends a token $(id, c_1, \ldots, c_k)$ to the TS (run by the challenger) together with a (blind) signature on it produced by the TS. Moreover, $\mathcal{A}$ provides $\gamma_1,\ldots,\gamma_k$ and $\nu_1,\ldots,\nu_k$. The TS then checks that these values are correct. If not, it aborts. So far, the challenger follows the protocol.

Now, we start to deviate from the protocol. The challenger looks up the corresponding tuple $(id, U, (i_1, \delta'_{i_1}, \kappa'_{i_1}), \ldots, (i_m, \delta'_{i_m}, \kappa'_{i_m}))$ from tokens it issued. Let $\overline{x_S}$ be the encrypted key share for this user. We use the values $\delta'_{i_1},\ldots,\delta'_{i_m}$ and $\kappa'_{i_1},\ldots,\kappa'_{i_m}$ to find the plaintext of one of $c_{i_1},\ldots,c_{i_m}$ and then use this to compute the plaintext of c_j.

For $i \in i_1,\ldots,i_m$ test if:

$$c_i = \overline{x_S} \cdot \mathbf{E}^+_{pk}(\delta'_i, \kappa'_i)$$

Let $(i^*, \delta'_{i^*}, \kappa'_{i^*})$ be the tuple that satisfies this equation. By, Lemma 1 we know that there must exist an index i^* such that:

$$c_{i^*} = \overline{x_S} \cdot \mathbf{E}^+_{pk}(\delta_{i^*}, \kappa_{i^*}),$$
$$\Delta_{i^*} = \mathsf{ExtCommit}((\delta_{i^*}, \kappa_{i^*}), \xi^*),$$

so this procedure does indeed find such a tuple $(i^*, \delta'_{i^*}, \kappa'_{i^*})$. The plaintext of c_{i^*} thus is $x_S + \delta'_{i^*}$. If $i^* = j$ we are done, and $\tilde{x}_S = x_S + \delta_{i^*} \pmod p$. Otherwise, the plaintext of c_j is $x_S + \delta_{i^*} + \gamma_{i^*}$ and therefore $\tilde{x}_S = x_S + \delta'_{i^*} + \gamma'_{i^*} \pmod p$.

Now that the challenger has derived $\tilde{x}_S$ it continues with the TCP as normal. This shows how we can answer RunTCP queries without needing to decrypt the ciphertexts.

Games 0 and 1 cannot be distinguished by the adversary. During ObtainKeyShareToken queries, the TS extracts the token identifier id using rewinding, so this is not detected by the adversary. By Lemma 1 the index i^* exists with overwhelming probability, so the responses of the TS are completely identical for the RunTCP queries made by the adversary.

In Game 2, we do not send $\overline{x_S} = \mathbf{E}^+_{pk}(x_{U^*,S})$ to the adversary when it makes RegisterUser queries for the challenge user U^*. Instead, we send $\overline{x_S} = \mathbf{E}^+_{pk}(0)$. During RunTCP queries, we first extract the plaintext of c_j as above, and then add $x_{U^*,S}$. The fact that the TS does not need to decrypt c_j to answer RunTCP queries together with Lemma 5 shows that the adversary cannot detect this change.

In Game 3, we again change how we answer RunTCP queries for the challenge user U^*. In particular, we will answer this query without using the corresponding key-share $x_{U^*,S}$. Instead, we use the challenge oracle for the TCP security in the query phase. We proceed as before, to find the plaintext δ of c_j when running GenShares. However, now we use the TCP challenge oracle to run the TCP by making a TCP(δ mod p) query. The TANDEM security challenger relays the messages to the adversary $\mathcal{A}$. After the selection phase, we advance the TCP security challenger to the challenge phase. Moreover, the challenge user U^* cannot obtain new tokens (because U^* is either blocked or rate-limited), and all old tokens have been invalidated, so we no longer need access to the TCP oracle to answer queries. Finally, in the challenge phase, we relay the messages to the TCP challenger. Then, if adversary $\mathcal{A}$ wins Game 3, it breaks the TCP security of the underlying TCP. Since we assumed this cannot happen, the TANDEM scheme is secure as well. The only difference between Game 2 and Game 3 is that we use the TCP oracle to run the TCP. However, since the TCP oracle uses to correct randomized key, this change is indistinguishable to the adversary. □

B.4 Privacy Proof

In our privacy proof, we reduce an attacker against privacy to an attacker on the underlying blind signature scheme (which we instantiate using the BBS+ credential scheme). In terms of attribute-based credentials, this game is precisely the issuer-unlinkability game. This game is the standard blind-signature game [JLO97; SU17].

GAME 6. The *blind-signature game* is between a challenger controlling an honest user U and an adversary controlling the signer.

Setup At the start of the game, $\mathcal{A}$ publishes the public key of the signer and outputs all other necessary public parameters.

Challenge At some point, the adversary outputs two messages m_0 and m_1 on which it wants to be challenged. The challenger picks a bit $b \in_R \{0, 1\}$ and proceeds as follows. (1) User U engages with the signer to obtain a signature on m_b. Let σ_b be the corresponding signature. (2) User U engages with the signer to obtain a signature on m_{1-b}. Let σ_{1-b} be the corresponding signature. If either signing protocol fails, set $\sigma_0 = \sigma_1 = \perp$. Finally, the challenger sends (m_0, σ_0) and (m_1, σ_1) to the adversary.

Guess Finally, the adversary outputs a guess b' of b. The adversary wins if $b' = b$.

If no adversary can win this game then the signer can recognize neither the signature nor the message.

The computationally hiding commitments in the ObtainKeyShareToken protocol ensure that the TS learns nothing about the unrevealed ciphertexts c_i which it then blindly signs—again without learning anything about the message. So, when the user runs GenShares and thereby reveals these ciphertexts, they cannot be directly correlated to a corresponding run of ObtainKeyShareToken. Moreover, the plaintext corresponding to the ciphertexts c_i are fully randomized, so that these too do not reveal anything about the user with which the TS is currently interacting.

The privacy proof folows a sequence of games. Throughout we use a guess i_0, i_1 for the challenge tokens. If this guess turns out to be incorrect when the adversary makes it challenge query, we abort and try again. We first use a sequence of games to show that we can remove identifying information from the ObtainKeyShareToken protocol.

- Game 0 is the Tandem privacy game, see Game 4 on page 55.
- In Game 1, we extract the TS key-shares $x_{0,S}$ and $x_{1,S}$ for users U_0 and U_1 from the TS' proof of knowledge in step I of the RegisterUser protocol, see Appendix B.2.
- In Game 2, we forge the user's zero-knowledge proof of correct construction of C, the commitment to the token identifier id and the randomized ciphertexts, at the end of ObtainKeyShareToken protocol.
- In Game 3, we replace the ciphertext $\overline{id}$ with the encryption of 0. The CPA security of the encryption scheme ensures that the adversary cannot detect this change.
- In Game 4, we use the extractability of $\mathsf{ExtCommit}(\cdot,\cdot)$ to forge the user's cut-and-choose proof in the ObtainKeyShareToken protocol, and send random commitments C_i, Δ_i for $i \notin \mathcal{D}$. However, we honestly construct C as per the protocol.
- In Game 5, for user U_i and the challenge token, we set $c_i = \mathbf{E}^+_{pk}(x_{i,S} + \delta_i, \kappa_i)$ for $i \notin \mathcal{D}$, rather than using $\overline{x_S}$. We commit to c_i for $i \notin \mathcal{D}$ as usual. Lemma 4 shows that with high probability we still follow the protocol correctly.
- In Game 6, we omit $x_{i,S}$ altogether in the construction of the unrevealed c_i, that is, we set:

$$c_i = \mathbf{E}^+_{pk}(\delta_i, \kappa_i) \tag{12}$$

  for all $i \notin \mathcal{D}$, and use these values to construct C. When answering RunTCP queries, user i adds $x_{i,S}$, which we extract during the RegisterUser protocol, to its long-term secret-share x_U to compensate for this change. The size of the randomizers δ_i ensure that the TS cannot detect this change.

We are now in the situation where the tokens held by user 0 and 1 are exchangeable. We use this to show that no adversary can distinguish situations $b = 0$ and $b = 1$. We use a sequence of games to interpolate between the two situations. We start from Game 6.

- In Game A, the challenger uses $b = 0$ but otherwise proceeds as in Game 6.

- In Game B, the challenger swaps the signatures of the challenge tokens of users U_0 and U_1. By the blind signature game, the adversary cannot detect this change.
- In Game C, the challenger also swaps the users U_0 and U_1 in the challenge phase. As a result, it perfectly simulates $b = 1$ in Game 6. The privacy property of the threshold cryptographic protocol (with colluding respectively honest SP) ensures that the adversary cannot detect this change.

Since these steps are indistinguishable, no adversary can distinguish the situations $b = 0$ and $b = 1$ in Game 6, and by indistinguishability again, neither can any adversary distinguish these two in the original privacy game.

Proof of Theorem 2. Throughout this proof, we use a guess for the challenge tokens i_0 and i_1 of users U_0 and U_1 respectively. If this guess turns out to be wrong in the challenge step, we abort and try again.

In Game 1, the challenger extracts $x_{0,S}$ and $x_{1,S}$ for users U_0 and U_1. In particular, the challenger runs the knowledge extractor on the proof of knowledge of the RegisterUser protocol, see Equation 7, for each of the users. Since the extractor uses rewinding, the adversary does not detect this.

In Game 2, the challenger forges the proof of knowledge of correctness of the commitment C at the end of the ObtainKeyShareToken protocol for the challenge tokens i_0 and i_1 of users U_0, U_1 respectively. By simulatability of this proof, the adversary cannot detect this change.

In Game 3, the challenger replaces the encryption of the token identifier *id* for the challenge tokens i_0 and i_1 with the encryption of the value 0. The proof of knowledge of correct encryption is already forged in the previous game. A reduction to the CPA security of the encryption scheme shows that an adversary that can distinguish Games 2 and 3 can break the CPA security of the encryption scheme.

In Game 4, the challenger extracts the subset $\mathcal{D}$ from the commitment Δ as soon as it receives it. For the two challenge tokens, the challenger (acting as the user) now proceeds as follows. It computes C_i, Δ_i for $i \in \mathcal{D}$ as per the protocol. However, for $i \notin \mathcal{D}$ it lets the unrevealed commitments C_i and Δ_i commit to random values. The proof of knowledge that C commits to the same values as C_i is already forged since a previous step. Because the commitment scheme is information theoretically hiding, the adversary cannot detect this change.

Despite the changes we made, the final token that is stored by the user is exactly the same as in the original ObtainKeyShareToken protocol. In Game 5 we compute the values c_i for user U_j and $i \notin \mathcal{D}$ as $c_i = \mathbf{E}^+_{pk}(x_{j,S} + \delta_i, \kappa_i)$ (recall, we extracted $x_{j,S}$ in the RegisterUser phase) instead of $c_i = \overline{x_S} \cdot \mathbf{E}^+_{pk}(\delta_i, \kappa_i)$. Lemma 4 shows that with overwhelming probability $\mathbf{D}^+_{sk}(\overline{x_S})$ equals the value $x_{j,S}$ we extracted in the RegisterUser protocol, so this change does not modify the adversary's view.

In Game 5, the user U_j computes

$$c_i = \mathbf{E}^+_{pk}(x_{j,S} + \delta_i, \kappa_i)$$

In Game 6, we remove the $x_{j,S}$ component from this equation, and instead just compute

$$c_i = \mathbf{E}^+_{pk}(\delta_i, \kappa_i) \tag{13}$$

for the challenge tokens. To compensate for the fact that $x_{j,S}$ is no longer included, the users adds $x_{j,S}$ to x_U. As a result, the threshold cryptographic protocol still completes as before.

The size of the domain from which the δ_is are drawn, ensures that the adversary cannot detect this change. More formally, the user sends c_is, γ_is, and v_is. However, the last two sets are redundant, they can be computed directly based on the c_is. As a result, we can focus on $\delta_i = \mathbf{D}^+_{sk}(c_i)$. By the size of the domain δ_is and the size of $x_{j,S}$ tuples $(\delta_1, \ldots, \delta_k)$ and $(x_{j,S}+\delta_1, \ldots, x_{j,S}+\delta_k)$ are statistically indistinguishable. As a result no adversary can distinguish Games 5 and 6.

We now show that no adversary can win Game 6. We again use a sequence of games, but now interpolate between Game A, where the challenger uses $b = 0$ in Game 6, and Game C, where the challenger uses $b = 1$ in Game 6. We construct the intermediate Game B, where user U_0 uses the token i_1 of user U_1 and vice versa. Since the challenge tokens in Game 6 (and thus in Games A, B, and C) do not depend on the user, the threshold-cryptographic protocols complete correctly as in Game 6.

We first show that Games A and B are indistinguishable. Suppose to the contrary that $\mathcal{A}$ can distinguish Games A and B. We show that we can use $\mathcal{A}$ to build an adversary $\mathcal{B}$ that breaks the blindness property of the signature scheme. In the blind signature game, $\mathcal{B}$ gets oracle access to two users that request a blind signature on one message each. Adversary $\mathcal{B}$ acts as the challenger towards $\mathcal{A}$ in Game 6. At the start of the game $\mathcal{B}$ generates two messages, corresponding to key-share tokens, for which users U_0 and U_1 need a blind signature. It creates:

$$m_0 = (id, H(c_1), \ldots, H(c_k)) \tag{14}$$

$$m_1 = (id', H(c'_1), \ldots, H(c'_k)), \tag{15}$$

where the values in the tuples are as in Game 6. Adversary $\mathcal{B}$ sends m_0, m_1 to its blind signature challenger.

During the ObtainKeyShareToken protocols for the challenge tokens, $\mathcal{B}$ simulates its users as follows. When user U_0 is running the blind signature protocol to create the challenge token τ_0, $\mathcal{B}$ uses the its challenger of the blind signature game to act as the user. When U_1 runs the blind signature protocol to create token τ_1, $\mathcal{B}$ again uses its blind signature game challenger. Finally, the blind signature challenger outputs two signatures σ_0 and σ_1 on messages m_0 and m_1 respectively. Adversary $\mathcal{B}$ uses σ_0 to construct the key-share token for user U_0, and uses σ_1 to construct the key-share token for user U_1. If $b = 0$ in the blind-signature game, $\mathcal{B}$s challenge user first blindly signed m_0, so $\mathcal{B}$ perfectly simulates Game A. If $b = 1$ in the blind-signature game, then $\mathcal{B}$ perfectly simulates Game B. Hence, any distinguisher between Games A and B breaks the blindness property of the blind signature scheme.

We now show that if the TCP scheme is private (with a colluding respectively honest SP), no adversary can distinguish between Games B and C. Suppose to the contrary that adversary $\mathcal{A}$ can distinguish Game B from Game C. We show that we can use $\mathcal{A}$ to build an adversary $\mathcal{B}$ that breaks the privacy property of the TCP scheme. Adversary $\mathcal{B}$ simulates users U_0 and U_1 towards $\mathcal{A}$. The RegisterUser and ObtainKeyShareToken protocols do not involve the users' secrets, so $\mathcal{B}$ computes them directly. We now show how to answer RunTCP queries.

Whenever $\mathcal{A}$ makes a $\mathsf{RunTCP}(U_i, j, \mathsf{in}_U)$ query, $\mathcal{B}$ makes a $\mathsf{RunTCP}(i, \mathsf{in}_U)$ query of its challenger. Distinguisher $\mathcal{B}$'s challenger replies with the TS' key-share $\tilde{x}_S$. Let $\tau = (\sigma, \epsilon, id, (c_l, \kappa_l, \delta_l)_{l=1,\ldots,k})$ be the jth token of user U_i. Normally, this token dictates a TS key-share unequal to $\tilde{x}_S$, but we can use the random oracle and

change the token to ensure that the TS will recover $\tilde{x}_S$. To do so, adversary $\mathcal{B}$ picks $\delta'_1, \ldots, \delta'_k \in_R \{0, \ldots, p2^\ell\}$. Let δ'_m be the largest, then we slightly increase this value (by at most p) so that $\delta'_m = \tilde{x}_S \pmod p$. (With overwhelming probability this modified δ'_m is less than 32^{ℓ_δ}; if not, we try again.) Then, we pick $\kappa'_1, \ldots, \kappa'_k \in_R \mathcal{R}$ and set $c'_l = \mathbf{E}^+_{pk}(\delta'_l; \kappa'_l)$. Adversary $\mathcal{B}$ updates the random oracle to ensure that $H(c'_i) = H(c_i)$, i.e., the new pairs have the same hash values as the original pairs. Next, $\mathcal{B}$ uses token $\tau' = (\sigma, id, (c'_l, \kappa'_i, \delta'_l)_{l=1,\ldots,k})$ to run GenShares with the TS. The changes to the random oracle ensure that this token is valid. Moreover, the changes to the random oracle succeed with high probability since at no point in the games does the TS learn the inputs to these hash-functions. The TS will derive the correct secret share $\tilde{x}_S$ from τ'. So it runs the correct TCP protocol with the requested user which is simulated by $\mathcal{B}$'s challenger.

To answer $\mathcal{A}$'s challenge queries, $\mathcal{B}$ again uses his challenger and proceeds as above to answer the queries. If $b = 0$ in the TCP privacy game, then $\mathcal{B}$'s first run of RunTCP uses user U_0's key, so $\mathcal{B}$ simulates Game B. Otherwise, if $b = 1$, then $\mathcal{B}$ simulates Game C. So, any adversary $\mathcal{A}$ that can distinguish Games B and C breaks the privacy property of the TCP scheme. This completes the privacy proof. □