Aspects of Secure and Efficient Streaming and Collaboration

GUNNAR KREITZ

KTH VETENSKAP OCH KONST

KTH Computer Science and Communication

Doctoral Thesis in Computer Science
Stockholm, Sweden 2011

KTH Computer Science
and Communication

Aspects of Secure and Efficient Streaming and Collaboration

GUNNAR KREITZ

Doctoral Thesis
Stockholm, Sweden 2011

TRITA-CSC-A 2011:05
ISSN 1653-5723
ISRN–KTH/CSC/A--11/05-SE
ISBN 978-91-7415-942-4

KTH
School of Computer Science and Communication
SE-100 44 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i datalogi fredagen den 13 maj 2011 klockan 13.00 i sal D2, Lindstedtsvägen 5, Kungl Tekniska högskolan, Stockholm.

Tryck: E-print

Abstract

Research within the area of cryptography constitutes the core of this thesis. In addition to cryptography, we also present results in peer-assisted streaming and web security. We present results on two specific cryptographic problems: broadcast encryption and secure multi-party computation. Broadcast encryption is the problem of efficiently and securely distributing content to a large and changing group of receivers. Secure multi-party computation is the subject of how a number of parties can collaborate securely. All in all, this thesis spans from systems work discussing the Spotify streaming system with millions of users, to more theoretic, foundational results.

Streaming is among the largest applications of the Internet today. On-demand streaming services allow users to consume the media content they want, at their convenience. With the large catalogs offered by many services, users can access a wide selection of content. Live streaming provides the means for corporations as well as individuals to broadcast to the world. The power of such broadcasts was shown in the recent (early 2011) revolts in Tunisia and Egypt, where protesters streamed live from demonstrations.

To stream media to a large global audience requires significant resources, in particular in terms of the bandwidth needed. One approach to reduce the requirements is to use peer-to-peer techniques, where clients assist in distributing the media. Spotify is a commercial music-on-demand streaming system, using peer-to-peer streaming. In this thesis, we discuss the Spotify protocol and measurements on its performance.

In many streaming systems, it is important to restrict access to content. One approach is to use cryptographic solutions from the area of broadcast encryption. Within this area, we present two results. The first is a protocol which improves the efficiency of previous systems at the cost of lowered security guarantees. The second contains lower-bound proofs, showing that early protocols in the subset cover framework are essentially optimal.

Many streaming systems are web-based, where the user accesses content in a web browser. Apart from this usage of the web, subscriptions for streaming services are bought using a web browser. This means that to provide a secure streaming service, we must understand web security. This thesis contains a result on a new type of attack, using an old history detection vulnerability to time the execution of a redirect of a victim's browser.

Within the area of secure multi-party computation, this thesis has three contributions. Firstly, we give efficient protocols for the basic functions of summation and disjunction which adapt to the network they run on. Secondly, we provide efficient protocols for sorting and aggregation, by using techniques from the area of sorting networks. Finally, we prove a dichotomy theorem, showing that all functions with three distinct outputs are either maximally easy or maximally difficult with regards to the security provided.

Foreword

During my Ph.D. studies, I have had the opportunity to work on a number of different topics. My main research interest has been cryptography, studying how we can cooperate with people we do not trust, and how to securely send information to large numbers of recipients. Apart from cryptography, I have also been interested in more general security issues, such as web security.

After having started my Ph.D. studies, I also begun working part-time at the Swedish company Spotify, which develops and runs a music streaming service. This lead me to study the issue of how to efficiently (quickly and cheaply) stream to a large number of users.

Organization of this Thesis

This thesis is split into two parts. The first part consists of an introduction to the topics contained within the thesis, and the second part consists of self-contained articles. The first part briefly summarizes and unifies the topics contained in the latter part, hopefully in an easy-to-read manner. The first chapter is intended to be accessible also for a non-expert reader.

As the thesis contains contributions in several topics and sub-fields, the first part also includes a somewhat more comprehensive scientific background on the topics. It is my hope that these summaries can be easily read by experts from other fields, and serve as a more detailed and gentle introduction to the field compared to the background sections of the articles.

The first part of the thesis is not intended to add new technical material beyond what is contained in the papers, but does contain a more elaborate discussion on the background and context of the results.

The second part contains seven papers, constituting the main scientific content of the thesis.

Contents

Acknowledgments

Well, well, well. Here I am, writing a thesis, summarizing the results of six years' worth of work. There are a number of people I would like to thank for their help, support, and friendship over these past years.

First and foremost, I would like to thank my advisor, Johan Håstad. With the right mix of listening, gently nudging in the right direction, and imparting of knowledge, every visit to Johan's room makes you come out with increased understanding. Johan's ability to tackle any problem in the diverse areas a Ph.D. student may happen to wander into is impressive.

The theoretical computer science group at KTH is a great group to work in and I want to thank my present and previous colleagues in the group. I would like to thank Pedro de Carvalho Gomes, Emma Enström, and Siavash Soleimanifard for the many great lunches, fun discussions and beautiful plans. Thank you to Douglas Wikström for the interesting and enlightening discussions. I also want to thank Torbjörn Granlund for his neighborly spirit.

I would especially like to thank my co-authors, co-founders, teammates and past co-inhabitants of room 1445, Per Austrin and Fredrik Niemelä, and my co-founder, coach, and twice colleague Mikael Goldmann. Working together with you in all our various projects over the years has been a huge amount of fun. I am happy to see that the spirit of room 1445 thrives with my office mates Lukáš Poláček and 黄桑霞 (Sangxia Huang), and has even spread to neighboring rooms.

Spotify is a fantastic place to work at. Being part of the company and seeing it grow from the first internal prototypes to the product and user base we have today has been a truly amazing journey to be part of. I would like to express my gratitude to all my colleagues at Spotify, past and present. You truly are a great group of people, and you are by now much too many to list by name.

A big thank you to Emma Enström, Johan Håstad, Frida Löfqvist, Mikael Goldmann, and Pehr Söderman for reading and commenting on draft versions of this thesis. Thank you to Lukáš Poláček who contributed the first three words of this chapter.

Last but not least, I want to express my gratitude to Frida Löfqvist. Without you, the past five years would not have been the wonderful experience they have been. Thank you for (among other things) listening, commenting, understanding, and always being there!

Publications Included

This thesis is based on, and includes, the following seven papers:

I Gunnar Kreitz and Fredrik Niemelä, "Spotify – Large scale, low latency, P2P music-on-demand streaming," in IEEE P2P'10 [69].

II Gunnar Kreitz, "A zero-one law for secure multi-party computation with ternary outputs," in Theory of Cryptography Conference (TCC) 2011 [66].[1]

III Kristján Valur Jónsson, Gunnar Kreitz, and Misbah Uddin, "Secure multi-party sorting and applications," in Applied Cryptography and Network Security (ACNS) 2011 — Industrial Track (no formal proceedings).

IV Gunnar Kreitz, Mads Dam, and Douglas Wikström, "Practical Private Information Aggregation in Large Networks," in Nordsec 2010 [68].

V Gunnar Kreitz, "Timing is everything — the Importance of history detection," *in submission.*

VI Per Austrin and Gunnar Kreitz, "Lower bounds for subset cover based broadcast encryption," in AfricaCrypt 2008 [6].

VII Mattias Johansson, Gunnar Kreitz, and Fredrik Lindholm, "Stateful subset cover," in Applied Cryptography and Network Security (ACNS) 2006 [59].

During my Ph.D. studies, I have also co-authored the following two papers on computer science education, which are not included in the thesis (apart from this mention):

- Emma Enström, Gunnar Kreitz, Fredrik Niemelä, and Viggo Kann, "Test-driven utbildning — strukturerad formativ examination," in NU 2010.
- Emma Enström, Gunnar Kreitz, Fredrik Niemelä, Pehr Söderman, and Viggo Kann, "Five Years With Kattis — Using an Automated Assessment System in Teaching," *in submission.*

[1]In this thesis, we include the full version [67]. The difference from the conference version is that all proofs are included.

Chapter 1

Introduction

With the advent of a global real-time communications network and the fast development of computer hardware, we have entered an age of very large-scale and global services on the Internet. The most outstanding example *du jour* may be that of Facebook with over 500 million active users world wide, a number likely to have reached 600 million by the time this thesis has been printed.

Facebook is not the only, or even the largest, company with such a staggering number of users; Google has more traffic than Facebook [3]. Aside from such giants, there now also exist a large number of systems with user numbers in the tens of millions. Developing and operating such large systems poses a huge number of interesting and difficult problems. These range from technical to legal, with technical challenges coming from many different areas of computer science.

One concern that immediately arises from high user numbers is that of efficiency. For a system receiving millions of queries per second, the system must be highly efficient in its handling of requests. Similarly, if a system is to stream music to millions of users, it must be efficient in how it handles and delivers data.

Another concern is that of security of the system. Any system connected to the Internet is likely to be subject to fully automated attack scans looking for security holes. A large-scale system, as we are interested in here, is also likely to be the victim of more targeted and focused attacks. Security is a many faceted problem, with attacks ranging from non-technical, such as tricking people (called social engineering), to mathematical attacks on the cryptography used. In this thesis, we mostly consider the technical aspects.

There are many other challenges involved in building a large-scale system, but the two we discussed above, *efficiency* and *security*, are the focus of this thesis. We now turn to the question of what task we want the system to solve. Current large Internet applications include search engines, social networks, instant messaging, streaming, and games.

In this thesis, we discuss the security and efficiency of systems for two tasks: *streaming* and *collaboration*. The first task, streaming, is an area that has significant usage world-wide, with a large number of streaming services available, both for audio and video. The largest of these is Youtube, which streams short video clips over the web. There also exist a number of other large streaming systems, including PPLive, Mog, Rhapsody, Spotify, and Voddler.

The second task that we investigate, collaboration, is more theoretical. Here, we discuss if, and how, people can cooperate without necessarily trusting all other participants. This is the field of *secure multi-party computation*, a fundamental problem in the area of cryptography. This is an area with beautiful theoretical results; we can jointly perform computations on secret data from multiple sources, with the data remaining secret. While the results from this area can enforce a strong notion of security, the constructions have so far seen relatively little practical use in the real world.

In this introductory chapter of the thesis, we attempt to gently introduce the reader to the thesis work, and the previous and related results in the areas discussed. We begin with a discussion on streaming, and some topics related to streaming systems. We then proceed with an introduction to the field of secure multi-party computation.

1.1 Streaming and Efficiency

One of the important Internet applications, that has seen tremendous growth over the past few years is *streaming*. Streaming in this context means to send media over a network in a way such that playback begins before all content has been downloaded. This is in contrast to downloading a media file, such as an mp3 file, and then playing it when the download is complete.

As broadband Internet access is becoming more prevalent, streaming content with a quality acceptable to most users has become feasible, and there is a large number of streaming services. The largest streaming service is currently Youtube, with its enormous library of (mostly) short video clips. While short, home-made video clips is a type of content unique to Internet streaming[1], there are also large streaming systems for more traditional types of content such as music, TV programs and full length movies.

The growth of streaming services adds load to the common Internet infrastructure. In North America during peak hours[2], over 40% of data downloaded is streaming media according to Sandvine [97]. The same report indicates that streaming also accounts for above 30% of peak traffic in Europe and Asia-Pacific. These high numbers underscore the importance of efficiency in streaming.

Where do we need to be efficient with the bandwidth? Is it the bandwidth of the service operator that we want to save? Is it the total bandwidth across the Internet, especially over long distances like transatlantic cables? Or are we concerned with the bandwidth consumption at the end user? All of these are potential bottlenecks,

[1] With the exception of TV shows such as America's Funniest Home Videos.

[2] Peak hours are the (3–5) hours of the day when the total bandwidth utilization is the greatest.

where it is imperative to be efficient. In our discussion here we restrict our attention to the bandwidth required by the service operator to run the service.

In this section, we discuss some questions pertinent to streaming. Firstly, we discuss what the content that we are streaming is, and claim that the type of content affects the technical design of a system. Secondly, we discuss how to efficiently disseminate the content, in particular using peer-to-peer streaming. Lastly, we touch upon the question of when to begin playing back streamed content. Having discussed these three topics related to efficiency in streaming protocols, we proceed to discuss issues related to their security in Section 1.2.

Types of Streaming

Within streaming, there are two main types of streaming considered today: *on-demand* and *live*. In live streaming, all receivers get the same content at (roughly) the same time, similar to a traditional radio or TV broadcast. A live streaming system may offer only one single channel, for instance live streaming the Olympics, a concert, or a specific radio channel resending its current broadcast over the Internet. Some live streaming systems offer a number of different channels or events being streamed, e.g., letting the viewer choose which game to watch.

In on-demand streaming, the streaming system lets the user select content from a large catalog. The user selects what to play, when to begin playing, and can then pause, skip within the stream, or switch to something else at any time. On-demand streaming is used for most types of media content, such as music, radio and TV programs, and movies.

Types of Content

Streaming protocols are being used today for streaming both audio and video. In live streaming, there is no large difference between the types of content streamed as audio and as video. In both categories we find sports events, conferences, and re-broadcasting a TV or radio channel on the Internet.

For on-demand streaming, the content types differ a bit more between audio and video. The most well-known type of content is probably video on-demand streaming. It encompasses a large span of different content types, ranging from short video clips to episodes of TV series and full-length movies. For on-demand audio streaming, the most prevalent use case is streaming music.

From a user's perspective, there is of course a difference between full-length movies and music tracks, but how large is the difference technically when streaming? Both types are encoded as streams of bits, with video having larger *bitrate* (number of bits per second of content). Traditionally, much of the research literature has focused on video streaming (live or on-demand) and, implicitly, the results have been assumed to apply to audio as well. As we discuss in Section 2.1 there are, however, important technical differences. A music-on-demand streaming system typically has a very large catalog where each individual track is short. In contrast,

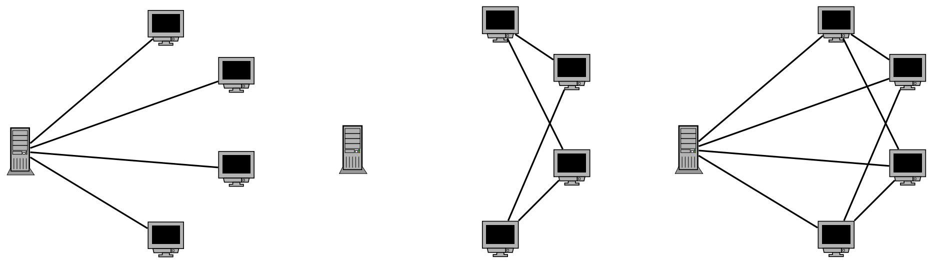

(a) Client-server (b) Peer-to-peer (c) Peer-assisted

Figure 1.1: Three types of protocols.

a TV or movie streaming service would have a much smaller catalog where each individual piece of content is large.

While there are important differences between different types of content, we write mostly as if considering a video service. This to avoid too cumbersome language. Thus, we write that a viewer watches a video.

Peer-to-Peer Streaming

Traditionally, the prevalent mode of operation across a network is the client-server paradigm. In such protocols every client connects directly to a server, which helps the client in solving a task. In the case of streaming, this means that each client individually streams content directly from a server. We illustrate this in Figure 1.1a. In streaming, a client-server protocol can provide good availability and response times, but is often costly to operate.

Peer-to-peer (P2P) protocols have been much discussed both in the popular press and academic literature. In a P2P protocol, the clients instead talk directly to each other to accomplish the task at hand, which we illustrate in Figure 1.1b. In most P2P protocols, a server remains in the picture, providing some functionality such as helping peers to find each other. These tasks are much simpler than the original problem (e.g., streaming). A useful property of a P2P system is that since each additional client helps in serving other clients, they more easily scale to large numbers of users. A drawback is that since content is sent from peers, it can be difficult to ensure that data is always available; the peers who had the data may have disconnected when a user wants it.

A combination approach also exists where a server provides the service, but P2P techniques are used to move parts of the load to the clients. This approach is known as *peer-assisted*, which we illustrate in Figure 1.1c. Peer-assisted protocols in many ways combine the best properties of client-server and peer-to-peer protocols. They can ensure availability and timeliness as in a client-server solution, but achieve much of the cost savings from a P2P system.

Protocols based on P2P have been proposed for a number of different problems. The most commonly studied and discussed task is that of file distribution, popularized with protocols such as Napster, Kazaa, Gnutella, and BitTorrent. File distribution protocols efficiently distribute a, typically large, file from one sender to those who wish to receive it. Distributing files is a very general mechanism, and it can be used for many purposes. Example applications include distributing a movie under a permissive license[3], and distributing software such as Linux distributions or World of Warcraft patches. It can also be used to distribute copyrighted material without a license. The latter usage typically gets more headlines than the former, and is sometimes equated with P2P in the popular press. There is much more to P2P and peer-assisted protocols than file sharing, however.

One example of a P2P protocol solving a different problem than file distribution is Skype, which is a protocol for Internet telephony or Voice over IP (VoIP). Another example is the subject of our present discussion, there is a large number of P2P streaming protocols, both for live and on-demand streaming.

In a live system, all users want the same content at (almost) the same time. Thus, a user can pass along all data she receives to a second user (or more). The second user can then forward the data to additional users, and so on. For an on-demand system, most content has been played at some time, so a user wanting to play a piece of content could stream not only from the service provider but also from other users who have previously played that same content, as they already have the material. This can significantly reduce the amount of content the original sender must transmit to the users of the system.

When to Begin Playing?

Having discussed how to distribute the data, we turn to another challenge in building a streaming system: when to begin playing. In a streaming system, the player receives data over a network. At some point in time it needs to start playing the data back to the viewer. When is the right time to begin playback?

A first idea may be "as soon as possible". There is a drawback with this idea, since network traffic can often vary over time. Sometimes, data sent over a network may be dropped, or there is a delay in its delivery, or a viewer's Internet connection may simply go down for a while. If the data does not arrive at a sufficiently fast rate, the client may not have the data it needs to play. This can either cause playback to pause, waiting for the data to be received, or to skip a bit of the video. This is known as a *buffer underrun* or *jitter*, we use the former term. By starting the playback immediately, there is a large risk of buffer underrun occurring.

To reduce the risk of a buffer underrun, a client typically stores received data in a buffer. It waits to commence playback until a sufficient amount of material has been received. How much is sufficient? As an extreme point, if it waits until

[3]Examples include the movies "The Yes Men Fix the World" and "Nasty old People", both released with a Creative Commons license and distributed using P2P.

the entire video has been buffered, then it can be certain to never suffer a buffer underrun. On the other hand, playback would take much too long to begin, and it would no longer even be a streaming system by our definition. A trade-off is involved: the larger buffer the client waits for before starting, the smaller the risk that a buffer underrun occurs during playback, but the longer the viewer has to wait before the video starts playing.

Creating a good mechanism to decide when playback commences is important in streaming systems, as it directly affects the user-perceived quality of the system. In Paper I we discuss the Spotify system and protocol. Spotify is a commercial music-on-demand service with millions of users. The system is rapid in beginning to stream, with a median time to begin playback of 265 ms. The frequency of buffer underruns is reasonably low, occurring in less than 1% of tracks played. This performance is achieved by a peer-assisted protocol and with a mechanism to decide when to begin playback that is based on a statistical construction known as a Markov chain.

Having finished our discussion on the efficiency of streaming systems, we now proceed to two topics related to their security.

1.2 Security in Streaming

There are many aspects of the security of an actual streaming system. We cannot hope to cover all of them, but we discuss two specific subjects: broadcast encryption and web security. Broadcast encryption is one approach to protect the content in a streaming system, and the primary motivation for studying broadcast encryption is content distribution.

Web security is a large topic, and the problems we discuss here are not specific to streaming. Web security is, however, an important topic in constructing a streaming system. Many streaming services are entirely web based, and it is thus important to understand the threats and solutions relevant to the web. Some services are not web based. Web security is still important to them as well as sensitive operations such as payment for subscriptions is typically handled over the web.

We begin with a discussion on broadcast encryption. Before this, we give a minimal background on basic cryptography.

Cryptography

In the field of cryptography, we study the foundations of how to protect information. The most classic example is that of two persons, Alice and Bob, who wish to send secret messages to each other even though Eve the Eavesdropper can see all messages they exchange. Alice and Bob can solve their problem by agreeing on a *secret key* and using a cipher to encrypt their communication. A key here is a large random number, sufficiently large that it cannot be guessed by an attacker. A cipher consists of a pair of methods, one for *encryption*, and the other for *decryption*. Encryption transforms a message and a key into a ciphertext. The decryption procedure takes as input a ciphertext together with the key used to encrypt it and gives back the original message. With a secure cipher, the message cannot be recovered from a ciphertext without knowledge of the key used to encrypt it.

To communicate, Alice can encrypt her message to Bob using the secret key they share. She sends the resulting ciphertext to Bob, who can decrypt it using the key and read the message. Eve can see the ciphertext, but as she does not know the key shared by Alice and Bob, she cannot in general recover the message.

The art of creating secure ciphers and breaking insecure ciphers has been a topic of study for several thousand years; one famous example is the successful attack on the Enigma cipher used by Germany in World War II. A more detailed discussion on the details of how a cipher works is beyond the scope of this thesis. There exists a number of ciphers which have been thoroughly analyzed and are believed to be secure, such as the Advanced Encryption Standard (AES) [86]. Omitting some details, we say that two persons who share a key can securely communicate with each other, even in the presence of eavesdroppers.

The security of the communication is based on the fact that Alice and Bob know a key which Eve does not know. But where do Alice and Bob get their key from? An easy case is if Alice and Bob can arrange a private meeting and decide upon a key, a key they can then use to secure their communication.

The ciphers we have discussed so far are called *symmetric*, as Alice and Bob must share the same key. Another option is the use of *asymmetric* cryptography where a key consists of two parts: a public key and a private key. These parts are such that messages encrypted using the public key can only be decrypted using the private key. They public key can, as the name implies, be published, and the private key needs to be kept secret. To communicate, Alice would encrypt messages to Bob using Bob's public key. The message could be decrypted by Bob, who knows his private key. Bob would then encrypt the response using Alice's public key.

There exist several methods for asymmetric cryptography which are believed to be secure, the most well-known and mostly used being RSA [94], named after Rivest, Shamir and Adleman. Another common asymmetric method is named after El Gamal [45]. Most of the cryptography in this thesis is of the classic symmetric type where Alice and Bob share a secret key.

Broadcast Encryption

What happens if it's not just Alice and Bob wanting to speak, but rather a streaming service that wants to send content to many millions of subscribers? When streaming data over a network it needs to be encrypted, as otherwise anyone who can eavesdrop on the network communication could receive the content without paying.

In this section, we consider a live streaming scenario, where all subscribers receive the same content at the same time. While there are other uses, this scenario is the case where applying broadcast encryption is most straightforward, and thus easiest to explain. Thus, the problem we consider is how to encrypt a single stream that is to be received by millions. From Section 1.1 we know that how to stream efficiently (without trying to protect the content) is a problem on its own. To focus the discussion on the new problem of protecting the content, we think of the simplest network possible: a single sender sends messages, and all messages sent are heard by all users. This is known as a *broadcast channel*.

Apart from the fact that the number of subscribers is large, a further difficulty is that it also keeps changing! With millions of subscribers, there are always new users signing up, and old users canceling subscriptions. The problem of how to protect content distribution to a large and dynamic group is known as *broadcast encryption*. When designing solutions, the sender is assumed to share a unique secret key with each individual user, allowing the sender to securely communicate individually with each user. The sender encrypts the message to the target user and sends it on the broadcast channel, so all users can see the ciphertext, but only the intended recipient can decrypt it. Such a key could be given automatically to the user's software the first time she buys a subscription, or be embedded in a hardware device such as a set-top box.

We begin our discussion with two very simple suggestions, which do not quite work. A first solution would be for the sender to encrypt the content to each subscriber individually. This solution is very inefficient as it means that the whole content stream must be individually encrypted and sent to the millions of subscribers. It would be much better if the sender only needed to encrypt and broadcast a single copy of the stream.

Encrypting the stream once, we could pick some fixed key, give it to all subscribers and use it to encrypt the stream. That would be an extremely efficient solution. The problem is that it cannot accommodate changes in the set of subscribers. If a user stops paying for her subscription, she could still remember the key and keep decrypting the stream. With this solution, a cheap user could sign up, receive the key, immediately cancel her subscription, and keep watching the stream for free.

Can we somehow encrypt and broadcast a single stream, but still handle subscribers leaving? Yes, if we sometimes change the key we use to encrypt the stream! As a number of cryptographic keys are involved, we give a name to the key we use to encrypt the media stream and call it the *media key*. A system may either change the media key every time the set of subscribers changes, or the changes may be done periodically. If we change the media key periodically, a user who cancels her subscription would lose the ability to decrypt the stream the next time the media key changes. Paying subscribers must receive the new key after it has changed, and how to efficiently distribute the new key is the central question in the area. Coming up with a mechanism to efficiently distribute a media key to the current subscribers (but no one else) solves the problem of broadcast encryption. We refer to such a mechanism as a *broadcast encryption scheme*.

A number of broadcast encryption schemes exist in the literature. An important property is whether a scheme is *stateless* or *stateful*. In both types of schemes each member is given a set of keys when she first joins. These keys are used to securely distribute updated media keys when the media key changes. But are the keys which are used to decrypt media keys ever updated themselves? If they are, we say that the scheme is stateful, and otherwise we say that the scheme is stateless.

The distinction between stateful and stateless is important if the scheme is used in a setting where a subscriber may miss data. Consider what happens if a subscriber's Internet connection goes down for a minute during which the media key is updated. As she then does not have the current media key, she cannot decrypt the stream until the next media key is distributed. In a stateful scheme, however, it is not certain that she will be able to decrypt the next media key. This happens if the messages she missed while offline contained updates to the keys she uses to recover the media key. Thus, a stateful scheme needs to be augmented with a mechanism for subscribers to recover if they missed any messages. It is better for a scheme to be stateless than to be stateful, but as we show in Paper VII one can gain considerable efficiency by making a stateless scheme stateful.

How can we construct a broadcast encryption scheme? We begin with what we here refer to as the *naive scheme*: each user is given a unique secret key which is also known to the sender. To send out a media key, the sender sends each subscriber a copy of the media key encrypted with the subscriber's key. This sounds quite similar to the two previous ideas we claimed to not quite work, what has changed? Now, we only send a single stream, encrypted with a changing media key. The media key is individually encrypted to each subscriber. This is much more efficient as cryptographic keys are much smaller than a media stream. While the sender still needs to encrypt a message individually to each subscriber, the messages are now at least small. But we can still improve significantly on this design.

Subset Cover

The most common way to design a broadcast encryption scheme is to use the subset cover paradigm [83]. Here, we keep the idea from the basic scheme that each user shares a secret key with the sender, but we now add keys shared between several users (a *subset* of users) and the sender. For instance, it may be that there is a key k known only to the sender (Alice), and the users Bob, Charlotte and David. In case all three of the users are current subscribers, k can be used to encrypt the media key, and we do not have to encrypt it to each of them individually. If one or more of them is not a member when a new media key is distributed, then we cannot use the key k (as it is known to a non-subscriber who could then decrypt the media key).

For each key, we consider the subset of users who know the key. If all users in the subset are currently members, the key is currently usable. As users subscribe and unsubscribe, keys (and their associated subsets) change back and forth between being usable and unusable. When the sender is to distribute a new media encryption key, she finds a number of subsets which together *cover* the current subscribers. By this we mean that each subscriber must be in at least one of the subsets used, and that no non-subscriber can be in one.

Having selected such a cover, the sender then encrypts the media encryption key with the key associated with each subset in the cover. This allows every subscriber to recover the media encryption key as she is in at least one of the used subsets and thus has the associated keys, and no non-subscriber can decrypt it as they do not have any of the keys used. The amount of data the sender needs to send (the *bandwidth overhead*) depends on the number of subsets she needs to use for the cover, so she wants this number to be as small as possible. Coming up with a good way of selecting the subsets of users who share keys is the most important part in designing a subset cover based broadcast encryption scheme.

In the worst case, each subset selected will cover only a single user. This means that we can never do worse than our naive scheme, but of course we would like to do better! One of the contributions of this thesis is a proof in Paper VI, showing that we cannot always do better. This means that no matter how smart we are in selecting the subsets, sometimes when there are few subscribers our scheme will perform the same as the naive scheme. We elaborate on this (and give the technical caveats) in Section 5.2.

Having discussed how to cryptographically protect the streamed content, we now turn to a quite different topic related to the security of a streaming system: web security.

Web Security

As more and more financial transaction and integrity-sensitive interactions happen over the web, it becomes more and more important to provide adequate security on the web. As a first layer of protection, we need to encrypt sensitive information such as credit card information as it is sent over a network. There are too many user and web-based stores on the Internet that it is impossible for every user to share a secret key with every store. Thus, to protect the communication between a user and a store, we turn to the asymmetric cryptography we mentioned earlier. Browsers have built-in support for sending information encrypted to a server, called Hypertext Transfer Protocol Secure (HTTPS).

Just like we avoided going into any details on how ciphers work, we gloss over the details of how HTTPS works. In the real world, there are some security concerns with HTTPS. These include security problems in earlier versions of the protocol[4], problems with identity management, and usability problems causing users to ignore warnings. To simplify our discussion, we assume that communication over HTTPS securely transfers information from a user to the intended destination.

Assuming (as we now have) that we have solved the problem of securely communicating information to and from a web browser by always using HTTPS, what remains to be discussed? A number of issues remain, both in how to build secure server software and security within the browser. We focus on the latter. Web browsers have evolved into complex pieces of software. For instance, web pages

[4]Most recently an issue with session renegotiation which was solved in RFC 5746 [93].

may include code written in the JavaScript programming language that the web browser needs to run. Web browsers also need to support a large number of formats and protocols. Sometimes, all this functionality can interact in unforeseeable ways, creating vulnerabilities that an attacker can use.

Within the context of security in a web browser, there is a large range of topics. For brevity, we focus on one specific security property that we want to enforce: we want to prevent a web site viewed in a browser from interfering with other sites the user visits or spying on the user. First, we briefly motivate this, and then we discuss a long-standing vulnerability that allows a web site to spy on its visitors in what has been perceived as a fairly harmless way.

Isolation Between Tabs

At the same time as a browser must support a large number of features, it must also offer strong protection and separation between content from different sources. Multiple web pages can be open at the same time, and it is expected that they cannot interfere with or spy on each other. As an example, a user should be able to make a purchase from a store, entering their credit card details, without any other page open at the same time being able to access that information.

In addition to different pages not being able to spy on each other, they should not be able to affect each other in other ways either. Being logged into a bank, a user can set up a transfer of money on the bank's web site. The fact that the user is logged in is tied to the browser. This is demonstrated by the fact that if the user opens a new tab and navigates to the bank's site, she will be logged in and can make transfers. We sketch how this works in Section 4.1. Clearly, we do not want a malicious page open in another tab at the same time as the user is logged in to her bank to be able to set up bank transfers from the user's account!

CSS History Detection

Apart from isolating pages open at the same time, there are also other concerns. A user should be able to expect some privacy from a browser. For instance, if a user visits a malicious web site, that site should not learn anything about what pages the user has previously visited, her *browsing history*. A site may use such information to target advertisements, for market research, or even blackmail[5].

An attack that allows a web site to test if a user has visited a specific site was publicly reported in 2000 [96], and was left open for a long time. Reasons for this include that the impact of the problem was perceived as being fairly minor, and it was difficult to fix. A solution was proposed in 2010 [8], which was implemented in Chrome and Safari. Recently (March 2011), the two largest browser vendors released new versions which also implement the fix, Internet Explorer 9 and Firefox

[5]Blackmail may seem far-fetched, but can be compared with "One Click Fraud" [35] where a fraudster displays a message to the user on an erotic web site asking them to pay a fee and threatening to send home an embarrassing "reminder" unless the user pays.

4. This leaves Opera as the only major browser still vulnerable in its latest version. We remark that Internet Explorer 9 is not compatible with Windows XP.

Most web browsers remember which links a user has visited. This information is used to color visited links differently from unvisited links, to provide assistance to a user navigating a web site. A malicious site can abuse the mechanism that links are shown differently depending on their status as visited or unvisited. The site `evil.com` may include a link to `secret.com`. When the user visits `evil.com`, the link will be rendered in one of two ways, depending on if the user has previously visited `secret.com` or not. In vulnerable browsers, the `evil.com` site can detect which way the link was rendered. The attack is called CSS history detection; Cascading Style Sheets (CSS) being the method by which a web site describes the graphical layout of elements, including how visited and unvisited links are styled.

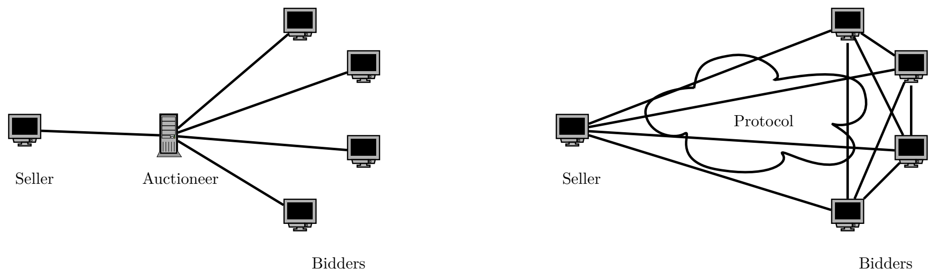
That a site can detect whether the user has visited a single link is less of a privacy invasion than being able to extract the full history of the victim's browser. If a malicious web site is able to test many links quickly, an attacker may extract a large fraction of the victim's history. In an optimized version, it has been shown that 30,000 links per second can be tested [54]. Some demonstration web sites were created to raise user awareness of the issue, both attempting to extract history of visiting popular sites [55] and specifically searching the user's history for erotic web sites [4]. These were published before the major browsers had implemented defenses against the attack.

In Paper V, we show that the impact of history detection attacks go beyond invasion of the victim's privacy by using the CSS history detection attack to time another attack. We show how a malicious page can use history detection to determine when a victim is about to pay for a purchase. By redirecting the victim's browser at an opportune moment, she can more easily be tricked into revealing credit card details or sending money to the attacker.

Having concluded our discussion on web security and the security of streaming systems, we proceed to the second topic of this thesis: how to securely and efficiently collaborate with people whom we do not fully trust.

1.3 Secure and Efficient Collaboration

The area of cryptography encompasses more topics than just protecting the security of communications, be they between two or more participants. One example of more complex tasks studied is that of secure voting. If we want to hold an election where votes are counted electronically, how can we do that securely? What do we even mean by securely? Clearly, each person should only be allowed to vote exactly once, and it should be impossible to know how an individual voted. We also want the voting system to protect against tampering; no organization should be able to alter the outcome of the election. Furthermore, we may want the election to be amenable to election monitoring by independent organizations who need to be able to verify that all votes were correctly counted. Independent observers of the election should



(a) Trusted Third Party (b) Secure multi-party computation

Figure 1.2: Replacing a trusted third party with a protocol.

not be allowed to learn how individual voters have voted.

Another related example of a more complex cryptographic task is that of auctions, and in particular sealed-bid auctions. A sealed-bid auction is one where all participants submit their bids in secret during a bidding period, and they are then all opened at once. In some settings, the losing bids remain secret even after the auction. Today, auctions are typically conducted through a middle-man, an auctioneer, whom the seller and bidders trust to honestly conduct the auction and maintain the secrecy of bids.

What do voting and auctions have in common? They are sensitive tasks where people want to cooperate in a well-defined way, but where the participants cannot fully trust each other. Today, such problems are typically solved by means of a *trusted third party*, i.e. a person or organization which can be reasonably trusted by all participants. In elections, we have election authorities and in auctions we have auctioneers. We illustrate a traditional auction with an auctioneer in Figure 1.2a. Sometimes there is no trusted third party available; in a corrupt state the election authorities may not be trusted, and in international high-profile auctions it may be impossible to find an auctioneer whom all parties trust. Can we somehow still have secure elections or auctions?

In secure multi-party computation, we study if and how we can replace a trusted third party with a cryptographic protocol. This is illustrated in Figure 1.2b. There, instead of communicating via an auctioneer, the seller and all bidders communicate using a cryptographic protocol. The goal is for the protocol to be just as secure as the solution with an honest auctioneer. The reader may compare Figure 1.2 to Figure 1.1 and note the similarities. In an analogy to the P2P paradigm, secure multi-party computation also investigates how a central function (server or trusted third party) can be replaced by a distributed protocol.

It turns out that in theory we can actually compute any function securely [46, 28, 15]! Furthermore, the security offered by these protocols is very good: no one learns anything but what they can deduce themselves from their input and the output of the function. What would those inputs and outputs be? In a common election, each voter inputs a single vote for a candidate, and the output is the number of votes each candidate received[6]. Thus, in a secure system, everyone learns the correct number of votes every candidate received, but learns nothing about who voted how. A voter should also be able to verify that her vote was correctly counted, a property that is difficult to impossible in current, real-world voting systems.

What are the inputs and outputs of an auction? First of all, what is an auction? Most readers will likely think of a traditional English auction where bidders raise their bids successively (often by shouting out or holding up a paddle). When no bidder wants to raise the bid, the auction is over and the last bidder wins, paying the price of the winning bid. There are, however, many different types of auctions studied. To simplify our discussion here, we will consider *closed* auctions, where bidders submit sealed bids (and thus, cannot interactively change their bids). To give a flavor of how auctions can differ from traditional auctions, we describe a Vickrey auction. In such an auction, the highest bidder wins, but only pays the bid of the second highest bidder. Compared to a normal auction, a Vickrey auction has the advantage that the best option for a bidder is to always bid the maximum price they are willing to pay, without any regard to how she believes other bidders will act. When creating a secure multi-party computation protocol, we can freely select the type of auction to implement.

Returning to our original question, what are the inputs and outputs of a closed auction? The input of the bidders is their bids, and the input of the seller is a reserve price. The outputs are more complicated, and here we have many options. The winning bidder should clearly learn that she won. Both seller and buyer need to know the final price. Should the seller learn the identity of the winner? In most auctions, yes, but in some cases, we may only want the seller to be able to communicate with the (anonymous) winner, to decide upon where and how to perform the transaction [102].

What should a losing bidder learn? She should learn that she did not win the auction. Should she learn what the winning bid was, the final price, or who won? Depending on the setting, the answer to each of these questions may be either yes or no. By using a secure multi-party computation protocol, we can create all types of auctions, and precisely select who should learn what.

Given that the results on how to compute any function securely date back to the late 1980's, what is there left to research? While those results are very general, the protocols can be too slow if run by a large number of parties or on complex functions (and voting or auctions are complex in this sense). One area of research is to look for improved general solutions, or general solutions which work in different settings (one such example is the network requirements discussed below). A second area is to create protocols for specific interesting functions, such as auctions. By specializing the protocol, it is often possible to get more efficient solutions. Finally,

[6]There are many different election systems that we could consider and implement securely, here we only pick a simple one.

a third line of research investigates the limits of secure computation.

In the case of protocols for specific problems, a large number of functions have been studied. These include the auctions and voting problems we have discussed, but also problems such as comparing gene sequences or running database queries[7]. In Paper III we give efficient protocols for the problem of sorting. A related protocol also presented in that paper can be used for some database queries, e.g., to compute the items which two databases have in common. Our Paper IV gives efficient protocols for basic problems such as summation and computing the maximum input. The novelty in our protocols lies in that they can run on (almost) any network, while most secure multi-party computation protocols are designed for a network where all participants can communicate directly with each other.

When positively stating that we can securely compute any function, we swept an important condition under the rug. Namely the fact that the protocols only remain secure as long as a too large number of the participants do not collude to break the security. How large can we make this "too large" number be? The answer is "it depends", but often we can make it quite high. We return to this subject in Chapter 3. For now we think of it as half of the participants, which is true for the so-called *information-theoretic honest-but-curious* setting (terms also explained in Chapter 3).

In our Paper II we prove results related to these limits. There, we show a so-called zero-one law for a certain class of functions, those with a single trit (a value that is 0, 1, or 2) as output. More precisely, we prove that each such function can either be securely computed no matter how many participants collude to break the security (the best possible case), or it cannot be securely computed without honest majority (the worst possible case). For general functions (without the restriction to three outputs), there are functions which lie between these two extreme points [30].

To place the role of secure multi-party computation, and cryptography in some context, we briefly discuss the topic of how to actually run such a protocol.

Running a Secure Computation

Given that we have a secure protocol for secure multi-party computation for some task we want to solve, how do we actually run it? A few protocols, e.g. our protocol for summation in Paper IV, are so simple that they could be run directly by humans with calculators, playing cards [40], or PEZ dispensers [7]. These are rare exceptions, in reality the protocols will be implemented and run as computer programs.

The typical setting to run a computation is that each party has her own computer on which she runs a program implementing the protocol. Each party enters her secret input on her own machine. The computers of all the parties are connected by a network to allow them to communicate with each other. Once all parties have

[7]Some of these problems are studied in their own area and typically not as part of secure multi-party computation. These include voting (mix-nets) and database queries (private information retrieval).

entered their inputs in the local machines, the protocol executes, and when the computation is complete, each party receives her output.

Does a secure multi-party computation protocol solve all our security problems? No! It, and cryptography in general, is more of a basic building block upon which we can build secure solutions. Without a secure protocol as a foundation, we could not implement the functionality securely, but even with the protocol, there are many other issues which need to be solved.

For the execution to be secure, it is important that not only the protocol is secure, but also that the implementation and computers are secure. In most cases, it is unlikely that each party has the expertise or resources required to implement the protocol herself or hire someone to do it. A more likely scenario is that all parties would actually run the same software implementing the protocol, or buy the software from one of a small number of vendors. If the implementation a party uses is insecure, her security may be compromised no matter how good the protocol is. If the execution is extremely valuable, each party could in theory implement the protocol herself and use software verification techniques to prove that the implementation is secure. Similarly to the issue of the implementation of the protocol is the computer it is running on. If the operating system has vulnerabilities, a party may again be compromised.

In the foregoing description, vendors implementing secure protocols seem similar to being a TTP. A key difference here is that each participant only needs to trust the vendor that she is buying from. There does not need to be a single vendor that all parties trust.

This concludes our discussion on secure multi-party computation. We end the present chapter with a brief summary of the contributions of this thesis.

1.4 Contributions

This thesis is based on the contents of seven papers. Out of these, the author believes that the two strongest scientific contributions are those contained in Paper II, proving a zero-one law for secure computation with ternary outputs, and Paper I, describing the peer-assisted streaming in the Spotify system.

In this thesis, we give both protocols and bounds for two cryptographic problems: broadcast encryption and secure multi-party computation. This means that we both discuss how to solve problems, and also prove that some specific problems cannot be solved. We also present contributions in streaming, detailing a large, commercial system for music-on-demand streaming and providing measurements showing how the system performs in the real world. Furthermore, we also describe a new attack within the context of web security, and propose a new security policy to prevent it and related attacks.

Each of the four remaining chapters of this thesis contains a more detailed summary of the respective contributions within the area discussed in the chapter.

Chapter 2

Streaming and Peer-to-Peer Streaming

A streaming system is a system for distributing and playing media, characterized by its ability to begin playing before the entirety of the content has been downloaded. The core problem is to get data to the receiver at a sufficient rate to allow interruption-free decoding and playback of the media stream. Streaming systems have become very popular, with high user numbers. How to construct efficient streaming systems is a rich area of research and there are a large number of different topics that need to be addressed when constructing a streaming system.

A streaming system may be streaming either audio or video, and as we argue in Section 2.1 there are important technical differences between the two cases. In most of our discussion, we treat generic streaming problems that apply to both types of streaming. To avoid too cumbersome language we write as if discussing video streaming services, saying that a viewer watches a video, in discussions pertaining to both audio and video.

Providing background on the problems in streaming, we discuss three central questions in turn: what are we streaming, how do we distribute the data, and how do we measure the results? We then summarize our contributions in streaming after which we end the chapter with some conclusions and lines of future work.

2.1 Types of Streaming Systems

We begin our discussion on what type of media content is streamed, and how it affects the properties of solutions. A number of different types of content have been discussed in the literature. It is well recognized that there is clear difference between live and on-demand streaming in protocol design. As we argue, there are also important differences for on-demand systems between music and movies, apart from the difference in bitrates. We discussed the different types of streaming in Section 1.1, here we recall the terminology and turn to a more technical discussion.

Live vs. On-Demand

One of the first distinctions is between live and on-demand streaming, which is a categorization based on the availability of the content, and on the amount of content offered. To exemplify the difference, there is a clear difference between a video repository such as Youtube and a live broadcast of the summer Olympics. On Youtube, the viewer can fully control playback of videos, whereas viewing the Olympics she can only see what is happening at the moment. Additionally, a user of the on-demand Youtube system has a huge number of different videos to choose from, whereas the live Olympics viewer would likely only have a small number of channels to select from, showing the current sports events and some studio discussions.

The technical challenges posed by the two settings have both many similarities and differences. We focus here on the differences. An on-demand system typically offers much more control of playback, allowing the user to play, pause, and skip. In addition, an on-demand system typically provides a much larger library of content for the user to select from. A challenge in a live system is that it cannot afford to do much, if any, pre-processing of the content delivered. An example of a similarity is the topic of when to begin playback that we discussed in Section 1.1.

What about the timing of delivery? For a live streaming system, one is often interested in minimizing the delay from the stream content being injected into the system and until the user plays the content back. Similarly, in an on-demand streaming system, one is interested in maintaining a low delay between when the user selects an object and when playback begins. A major difference here is that in live streaming, users want the same content at the same time, while in on-demand system, all users are at a different point of playback.

How important is playback latency in an on-demand system? We are not aware of any studies measuring this directly. An indication that it may be important is given by Brutlag [21], who demonstrated the importance of speed in web browsers. In an experiment, they added 50–400 ms of artificial latency to Google web searches. Their results indicated that even such small additions of latency resulted in decreased usage, with an average -0.59% daily Google usage from the users with the most latency added.

The distinction between on-demand and live streaming is made in most of the existing streaming literature. Most systems and proposals are specifically target at one of the two use cases, but some systems are designed to handle both, such as PPLive [51].

Music vs. Movies

A distinction that is not made very often in the existing academic literature is that between music and movie streaming. Almost all of the existing literature is primarily concerned with video streaming of longer videos such as TV episodes or movies. We argue that in on-demand streaming, there are actually significant differences between streaming music and movies.

At a glance, it may seem that streaming music is simply easier than streaming video as the bandwidth involved is lower. As an example, the Spotify on-demand music services streams with an average bitrate of 96–320 kilobits per second (kbps), with most tracks being streamed using a variable bit rate (VBR) coding averaging 160 kbps. This can be compared with video streaming services where typical bitrates are 400–800 kbps. Recently, Liu *et al.* [74] presented the Novasky video streaming system, which streams at bitrates of 1–2 Mbps.

Locally caching previously played content appears to be much more useful in a music streaming service. In the Spotify service, our measurements show that 55.4% of the data used by the clients came from the cache on the local machine. While we are not aware of numbers on cache utilization in video streaming services, we believe them to be significantly lower; users are more likely to listen to the same music many times than they are to view the same movie many times. As videos are larger than music tracks, a cache in a music streaming systems can store many more objects than a cache of the same size in a video streaming system.

Another difference between the two types of systems is the amount of content available. Many of the commercially deployed music-on-demand streaming systems have catalogs with over 10 million tracks available for streaming [101, 79]. In movie streaming services, catalog sizes are several orders of magnitude smaller, the movie streaming site Voddler has a catalog of over 2,500 movies [104].

There is also a large difference in the play time of the items in the two systems. A full-length movie is about two hours long, while a music track is a few minutes. This means that users of a music streaming system start playback of many more objects than users of a movie steaming system. In addition, it appears that users of music streaming systems are highly active in selecting songs: our Paper I shows that 39% of playbacks in the Spotify system begin due to user selection (as opposed to continuing to the next scheduled track).

Taken together, this means that a music system must be efficient in streaming tracks from a large catalog. As new tracks begin playing often, it is particularly important that the system is efficient in beginning to play a new track. For systems using a distribution method based on peer-to-peer techniques (one of the most common designs, see Section 2.2), this means that a music streaming system must be good at quickly finding peers with the correct content, and efficient in rapidly setting up streaming from those peers.

At the same time, the lower bitrate and smaller pieces of content in a music streaming service implies that the actual download strategy used when streaming is less important compared to a video streaming system. For instance, the Spotify system uses a straightforward strategy downloading content in-order to good results. By in-order, we mean that the data is downloaded in the order it is needed for playback. In comparison, the mechanism to download content is a key piece of a peer-to-peer video streaming system.

Why would a peer-to-peer video streaming system not download content in-order? As typical cache sizes mean that a user can only store a few videos, it is important that the user quickly receives pieces of content which can be usefully redistributed to other users. When downloading in-order, all clients receive the same content in the same order. This leads to a situation where the beginning of videos are much more well-spread among peers than the last parts of videos, as viewers can switch videos without viewing the whole video. This means that there is an increased risk that the last part is lost from the peer-to-peer network.

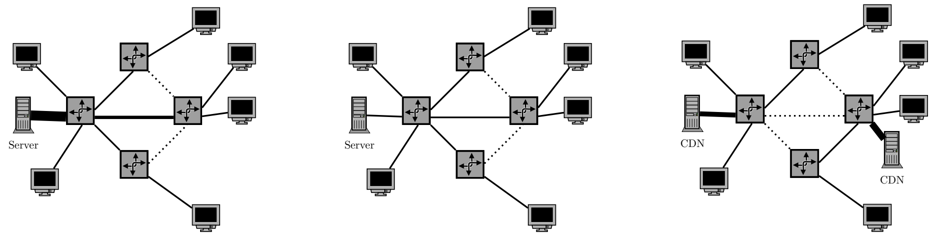
If each peer uses a more advanced strategy for what to download than in-order, the whole video can be quickly disseminated among the peers, and is thus more likely to remain available even after some peers have disconnected. This is similar to the BitTorrent [37] protocol for file distribution, and several proposed video streaming systems essentially use a modified BitTorrent protocol for streaming. But in a streaming system, data is needed in-order for playback, so the client cannot download all data out-of-order. This means that a video streaming system must strike a balance between downloading enough data in-order that is available for playback, while performing out-of-order downloading to achieve good data distribution properties. In a music streaming system, caches are comparatively large, so it is less imperative that a user can contribute upload capacity for the track they are currently playing. This observation is used in the Spotify protocol, where users only upload tracks which they have completely downloaded. This design simplifies the protocol and reduces protocol overhead.

Is the distinction truly between music and movie streaming? No, not quite. Systems with a huge catalog of short video clips, such as Youtube, have most of the challenges present in both movie and music streaming: the higher bitrate of video, but the large catalog and short playback times of music. We believe, however, that making a distinction between the two cases, and referring to them as music and movie streaming respectively immediately gives the right idea, even if there are also interesting systems “in between”.

2.2 Distribution Methods for Streaming Systems

A streaming system typically starts with a single sender who has some content that a number of users wish to receive. How is that content distributed over the network from the sender to the viewers? Consider a system with ten million concurrent users, streaming at 500 kbps. If the streams are naively sent from the server to the end users, this would use 5 Tbps of bandwidth at the server. This number can be compared to the DE-CIX Frankfurt Internet exchange point where the historical traffic peak at the time of writing is 3.2 Tbps [39]. There is clearly a need to devise a better distribution strategy than naively sending data to each user individually from a single point. In Figure 2.1a we illustrate a network where all clients connect directly to the server to download their content.

Here, we discuss three approaches to solving the data distribution problem. Firstly, we discuss multicast, which requires support for the underlying network, but results in optimal distribution for live streaming. Secondly, we discuss a solution



(a) Single server, unicast (b) Single server, multicast (c) CDN

Figure 2.1: Three modes of distribution. Thickness indicates bandwidth usage. Dotted indicates unused connection. Boxes with arrows represent network routers.

based upon what amounts to placing servers at many locations in the network, through the use of a content distribution network. Lastly, we discuss peer-to-peer based approaches. Both of the two latter approaches are practically usable today for both live and on-demand streaming.

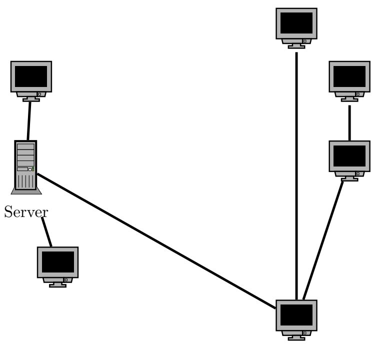
Multicast

The most common type of network traffic is *unicast*, meaning that a single sender sends data to a single recipient. Almost all traffic on the Internet is unicast. Another type of traffic is *broadcast*, where a sender can send a single piece of data that is propagated in the network to reach everyone in an addressed part (we omit details on what parts of the network can be addressed) of the network. Broadcast traffic is often targeted at the local network the sender is connected to. Due to security concerns[1], broadcast packets to networks beyond the local network are typically blocked.

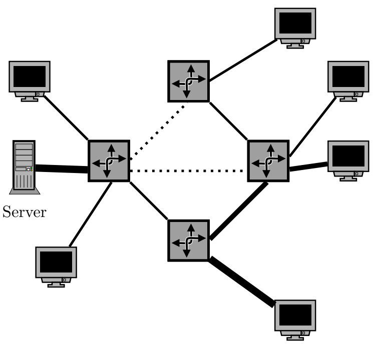
The Internet also includes a more complex mechanism called *multicast*. With this mechanism, computers can sign up, using the Internet Group Management Protocol [24] (IGMP) to receive traffic destined to a *multicast group*. A sender can then send a single packet addressed to the multicast group and have the routers in the network distribute the packet to all recipients in the group. The routers maintain a spanning tree from the sender to all receivers, so the data distribution is optimal for the network: each packet is only sent once over precisely the links required to reach all receivers. We illustrate this in Figure 2.1b.

While it may be difficult to apply multicast to on-demand streaming, it seems ideally suited to live streaming. Sadly, it is difficult to use multicast over the Internet. Due to several reasons, including scalability and network management

[1]An example of which is the Smurf attack [25], a classic Denial of Service attack where an attacker sends a ping packet to the broadcast address of a network in order to generate large amounts of network traffic.



(a) Spanning tree of unicast links



(b) Corresponding traffic on real network

Figure 2.2: A spanning tree overlay used for streaming, and the bandwidth utilization in the substrate network.

concerns, multicast is typically not deployed between Internet Service Providers, making it unusable in an Internet-wide streaming service. Multicast-based streaming can be a usable option for more local live streaming services to efficiently distribute information within a single Internet Service Providers' network.

The fact that multicast is not widely deployed, despite having been a part of the standard Internet protocols for over a decade was discussed by Chu, Rao, and Zhang [36]. They proposed to instead implement multicast on top of normal unicast links, building a peer-to-peer network. We illustrate this in Figure 2.2, where Figure 2.2a shows the structure of the virtual spanning tree that the parties are connected in. We show how this corresponds to network traffic in Figure 2.2b. We return to this construction in our discussion of peer-to-peer streaming later in this section.

Content Distribution Networks and HTTP-based Streaming

As it is difficult for a streaming service provider to distribute the huge data volumes involved, one strategy is to outsource the problem. Content Distribution Networks (CDNs) are businesses specialized in global content distribution. Among the large CDNs active today are Akamai, Amazon, and Level3.

Somewhat simplified, a CDN operates by having a large number of servers in data centers distributed across the globe and by intelligently distributing popular content to a data center near the users where the content is popular. Users are automatically directed to download content from the closest data center where it resides. This is shown in Figure 2.1c.

Distribution of content from the CDN to the end user uses unicast traffic, but in the common case over short distances. In addition, not all content is sent from the same source, meaning that there is no single hotspot needing tremendous amounts of bandwidth. Access to files hosted on a CDN is most often done using the Hypertext Transfer Protocol (HTTP) used for web browsing. Furthermore, by having many data centers, a CDN is able to offer good redundancy; if a user's closest data center has service problems, she can temporarily be redirected to another one.

As CDNs mostly use the standardized HTTP protocol for streaming, a service may opt to host parts of, or all of its material itself. It may then move content to a CDN as the need arises to save on server resources or bandwidth. Or alternatively, some large services such as Youtube essentially have their own CDN.

Most of the commercial streaming services today use CDNs for streaming. To the best of our knowledge, Spotify is the only music streaming service today not using a CDN (or self-hosted HTTP-based streaming) as its primary distribution mechanism. Within video streaming, the market is more mixed, with a few services such as PPLive, UUSee, and Voddler not building on CDNs.

Peer-to-Peer and Peer-Assisted

Peer-to-peer (P2P) systems have been intensely studied and garnered much attention in the popular press. While there are many uses for P2P techniques, here we are interested in their use in streaming. A large amount of academic research has been devoted to P2P-based streaming, and a number of streaming systems, both commercial and academic, based on P2P techniques have been deployed.

In a P2P streaming system, once a client has downloaded a piece of content, it may upload it to other clients. This way, a part of the bandwidth cost in the streaming service is moved from the sender to the receivers. A key point of P2P based systems is the scalability properties: as the number of users of the system increases, so does the serving capacity! P2P systems were largely popularized through P2P-based file-sharing services such as BitTorrent, Gnutella, and Napster.

Building a streaming system using P2P techniques, no special requirements are placed on the network, so it is easier to deploy than a multicast based solution. The peers make unicast connections to each other, sending and receiving data. Looking at the connections the peers make, we can imagine these as being a new, virtual network, running over the real, physical network. This network of the peers and their unicast connections is known as an *overlay* network, as it is running on top of another network. The underlying physical network is known as the *substrate*. This is illustrated in in Figure 2.2b.

What is the structure of this overlay network? At a minimum, we most often want it to be connected, so the simplest form is a tree. Some proposals do use a tree structure, or multiple overlapping trees, as we discuss below. Another common design approach is an unstructured, or *mesh*, overlay where peers connect to each other without any specific structure imposed.

To provide some background on design topics relevant to a peer-to-peer streaming system, we mention two fundamental design decisions: the structure of the overlay and basing distribution on pushing or pulling content. We also touch upon the distinction between peer-to-peer and peer-assisted systems.

Tree-Based and Mesh-Based

There are two primary approaches to structuring a P2P-based streaming system [77]: tree or mesh. In a basic tree-based system, a tree is formed with the sender at the root and the peers as nodes in the tree. Content propagates down in a tree along its edges; a peer sends content to its children in the tree. This approach is illustrated in Figure 2.2. Single-tree approaches include PeerCast [41], SpreadIT [10], and ZIGZAG [103]. Single-tree streaming systems can be understood as constructing a multicast tree on top of unicast links.

A problem with the single-tree approaches is that peers who end up as leafs in the tree do not contribute any bandwidth. Another issue is that a node near the top of the tree disconnecting can affect the quality for a large number of other users. When a user disconnects, some mechanism must be applied to repair the tree, and while the reparation is done, users below the disconnected user may not receive content.

One approach to reducing the impact of users disconnecting is through the use of multiple trees, as in CoopNet [88]. Tree-based approaches are typically targeted at the live streaming scenario where a large number of viewers want the same content at the same time. We are not aware of any tree-based proposals for on-demand streaming.

Mesh-based, or unstructured, approaches do not impose any particular structure on who talks to whom in the P2P network. Mesh-based approaches include Bullet [65], CoolStreaming/DONet [110], AnySee [73], PPLive [51], and Spotify [69] (Paper I).

Push-Based and Pull-Based

Another distinction in the literature is between push-based and pull-based approaches. In a push-based approach, peers distribute data to each other automatically, where in a pull-based approach a peer explicitly requests the data it needs. Similarly to the issue of tree-based and mesh-based design, the question of push vs. pull is mostly relevant in the context of live streaming [90] where many peers want the same content at the same time.

Push is mainly used in conjunction with tree-based system (and not in mesh-based systems), while pull is the most common paradigm in mesh-based streaming. Some approaches to live streaming built on *gossiping* have also been proposed. Gossiping is an important design paradigm in the field of distributed algorithms in general. For streaming, the approaches we are aware of are mesh-based and use gossiping to push information on what blocks are available at what peers, but then use a pull-based where peers ask someone they know to have data to send it to them. An example of a gossip-based system is Gossip++ [44].

Peer-to-Peer vs. Peer-Assisted

So far, we have called the systems we have discussed peer-to-peer. In many of the settings we consider, there is a distinguished node, a server, from which the streaming data originates, and the clients then only assist in offloading the server in data distribution. A purist may argue that such a system is not truly peer-to-peer as not all nodes are equal (peers). As was noted in e.g. [95], many popular deployed peer-to-peer systems also include some central part, such as the tracker in BitTorrent.

To be a bit more careful with terminology, the term *peer-assisted* is often used to describe systems where peer-to-peer techniques are used to augment or offload, rather than replace a client-server solution. Many of the protocols we have discussed are peer-assisted. An example of a more pure peer-to-peer streaming system is Tribler [91, 80].

In a peer-assisted system, an important design point is how a client is best to combine streaming from the two sources available: peers and the server. The server has better availability and bandwidth, and is typically guaranteed to be ready to serve all content. On the other hand, to achieve good offloading properties, clients should download as much as possible from peers. In the Spotify system, latency-critical requests (such as when the user clicks on a song she does not have in her cache) are sent to the server, while other requests are sent to the peer-to-peer network.

This concludes our overview of the basic design decisions on data distribution in streaming. We proceed to the related topic of how to measure the performance of a streaming system.

2.3 Evaluation of a Streaming System

There are several different measures of the efficiency of a streaming system. One measure is the amount of local resources in terms of storage, memory, or CPU are required from the sender, receivers, or both. Another measure of efficiency is the bandwidth required, where we could measure at the sender, receivers, some network links, or a combination thereof. In this thesis we are primarily interested in the bandwidth required at the sender. Focusing on the sender can be motivated by the fact that the resources required from the receivers are typically such that they are “reasonable”. Bandwidth is arguably the most important metric, and the one most directly affected by protocol design.

Apart from being interested in efficiency in the form of low resource consumption, we also want our streaming system to work well. What do we measure to know how well a streaming system works? One value of importance is the playback latency, the time taken from when the user clicks play until media playback begins. Another important parameter is how often playback is affected by a lack of data available at the client during playback. As throughput over a network varies with time, it may be that data needed for playback has not arrived at the client. Such events are referred to as buffer underruns, as mentioned in Section 1.1. The longer the client waits with playing, the more data it has when it starts playing, thus reducing the risk of a buffer underrun. This means that there is an inherent trade-off between playback latency and the frequency of buffer underruns, as discussed by Liang and Liang [72].

Given that there is a trade-off a system designer must select a target for her system. Is it better to start very rapidly, but have a large risk of buffer underruns, or to be more conservative? We are not aware of any user studies comparing different trade-offs. When evaluating the PPLive system, Huang *et al.* [51] proposed a metric called user satisfaction index weighing waiting time and the quality of playback (which is related to buffer underruns).

In the Spotify system, the trade-off has been selected such that at most 1% of playbacks should suffer from a buffer underrun. To do this, each client accumulates information on how the local network behaves as a Markov chain and uses this to simulate a playback of the song. If the simulations indicate that the risk for a buffer underrun is greater than 1% the client waits to commence playback. The approach is more fully described in Paper I.

2.4 Contributions

The contributions in the area of efficient streaming were published in Paper I. During his thesis work, the author has worked at the Swedish company Spotify which develops and operates a peer-assisted music on-demand streaming system. The peer-to-peer protocol is a pull-based, mesh-based structure. At Spotify the author has worked on designing the protocol used by the Spotify client, as well as other contributions to the Spotify system.

The peer-to-peer protocol designed and developed for Spotify is described in Paper I. As it is a large and deployed system, the efficiency of the protocol was demonstrated by measurements based on instrumentation of Spotify clients. At the time of evaluation, Spotify had over 7 million users. We show that the caching by clients and the peer-to-peer distribution mechanisms achieve significant resource savings: less than 10% of the data is sent from Spotify's servers. We also show that this efficiency is achieved without sacrificing the playback latency: the median playback latency is 265 ms, and 75% of played tracks start within 515 ms. The bandwidth efficiency is comparable to that of video-on-demand streaming in PPLive, where [51] claims that 7–11% of the data comes from the server.

2.5 Conclusions and Future Work

In conclusion, peer-assisted streaming protocols are used in practice today in large systems. There is still, however, room for future improvements. Firstly, we believe there is much room for future studies specifically targeting the music on-demand problem in peer-to-peer or peer-assisted settings. Much research has been devoted to video streaming, but as discussed in Section 2.1, music on-demand streaming has some quite different properties from movies on-demand streaming.

One open question is how to best select the trade-off between latency and the risk of buffer underruns. This would likely have to be empirically evaluated with users, as it comes down to what users dislike the most: interruptions during playback, or having to wait for the playback to commence. We would expect that good trade-off points may differ between different types of streaming.

Another open question in general on-demand streaming is how to provide a low-latency playback experience when streaming to peers. The Spotify system achieves low latency by always sending latency-critical requests to the central servers, but if some of these requests could instead be routed to peers, more bandwidth resources could be saved.

Chapter 3

Secure Multi-Party Computation

In Secure Multi-party Computation (MPC), we study how cooperation can be achieved without trust. In particular, we consider the task of n parties jointly evaluating a function $f(x_1, \ldots, x_n)$ where party P_i has a secret input x_i, a problem known as Secure Function Evaluation. The goal is that no party should learn anything beyond what they can deduce from their input and the output of the function. Furthermore, they should not be able to affect the output of the function, apart from the selection of their input.

MPC is one of the central areas of modern cryptography, and has been extensively studied since introduced by Yao [107]. The problem discussed by Yao was the *Millionaires' Problem*, where two millionaires want to learn who of them is the richest. They do not, however, want to reveal any other information about their wealth to each other.

The story of the two millionaires may seem a bit contrived, but could easily be turned into a more useful example, for instance as a first step in a price negotiation. Let our first “millionaire” be a seller, and the second “millionaire” be a potential buyer. The seller would put the absolutely lowest price she would be willing to sell at as her “wealth”, and the buyer the highest price he is willing to pay as his “wealth”. If the seller is “richest”, there is no way to find a deal, and no point in negotiating for a price. Otherwise, there is some price range to which both parties would be willing to agree, and it would be worthwhile to proceed. In this example, we also see why it may be important that no information leaks; if the seller were to learn the buyer's maximum bid, she would ask for that and refuse to go any lower.

Just as with the problems of voting and elections discussed in Section 1.3, one way for the two millionaires to solve their problem would be to find a trusted third party (TTP). They could then each reveal to the TTP how much money they have, and the TTP could tell them who has the most money. This solution generalizes; if we can find a TTP, we can securely evaluate any function by having all parties reveal their respective secret inputs to the TTP. Our goal is to construct a protocol solving the problem without a TTP.

Before discussing how to achieve our goal of replacing a TTP, we bring up another example. Consider the market for music subscription services. How large is the total value of the music streaming market? Each company knows its own revenue, but does not know the revenues of its competitors. We can view this as an MPC problem, where we wish to compute the sum of the companies' revenues. In this example, multiple parties participate in the companies, and not only two parties as in the millionaires' problem. More formally:

Example 3.1 *There are n companies active in the music streaming market. Each company P_i knows its last annual revenue x_i. The companies would now like to learn the total size of the market, $\sum_{i=1}^{n} x_i$, without anyone learning any additional information.*

As it turns out, summation is among the simplest functions to compute securely. As the example illustrates, there are also immediate applications for a protocol to compute the sum of inputs.

We now provide some background on secure multi-party computation. First, we discuss what it is we want to accomplish. Here, we discuss what our goals are with regards to security, and what we require from the network our protocols are run on. We then summarize the classic results showing that all functions can be securely computed. Then, we turn to the topic of understanding what cannot be done, discussing impossibility results and give a flavor of the most common proof technique. Next, we summarize our contributions, and discuss the real-world applicability of those, and MPC in general. Finally, we conclude and outline some future work.

3.1 Security Model

What do we mean by security in the context of MPC, and what can we hope to achieve? What we are trying to accomplish is not only to protect against outsiders, but to protect against attacks from the other parties participating in the computation. Intuitively, the goal is to come up with a protocol that performs the role of a TTP. This also means that we do not attempt to do anything more than what a TTP does. In particular, we cannot prevent a dishonest party from carefully selecting, or lying about her input.

Returning to Example 3.1, we do not prevent a dishonest company P_i from participating in the computation as if its input was $x_i' = x_i + k$, where k is some constant known only to P_i. The total size of the market that all parties would learn after the protocol execution would then be $k + \sum_{i=1}^{n} x_i$. Only company P_i would know to subtract k from the result, learning the true size of the market. The other companies would receive the wrong value as output, and not know. We do not consider this a security problem in an MPC protocol, as P_i could do the same if a TTP was performing the computation.

Attacks such as this could be somewhat mitigated by adding sanity checks to the inputs, e.g., if all companies are known to have roughly equal revenues, the function may be specified to forbid any inputs differing too much from other inputs. This would not prevent a party from lying, but would reduce the impact on the output of someone lying.

One approach to convince the parties to use truthful inputs may be to apply techniques from the area of algorithmic mechanism design [85], an area in the intersection of computer science and economic game theory. A more full discussion on this subject is beyond the scope of this thesis.

We now proceed to discuss the security we want to achieve. Firstly, we consider how parties may cheat, distinguishing between active and passive attacks. Next, we treat the case when two or more parties *collude* to break the security. Lastly, we discuss how computationally powerful the cheating parties are.

Passive vs. Active

Discussing security models, we begin with the question of how maliciously cheating parties are willing to behave. A number of models have been discussed, but here we consider the two most prominent models of what a cheating party is willing to do. These models are known as a *passive* or *active* adversary.

In the *passive* case, also known as honest-but-curious or semi-honest, cheating parties follow the protocol specification exactly, but record all messages they send and received in the protocol. After the protocol execution, they try to use this information to deduce information about the inputs of other parties. Their goal is to learn information that they could not deduce from their own inputs, and the output of the function. If the adversary cannot succeed with this, we say the protocol is secure.

In Example 3.1, a company can compute the sum of all other companies' revenues from its own input and the output of the function. Thus, this is not a violation of the security. It cannot, however, learn anything that cannot be inferred from the sum[1] about the individual inputs of other companies, or of a group of companies.

In the *active* case, also known as byzantine or malicious, a cheating party may misbehave arbitrarily. She may send whatever messages she wants, at whatever time she wants, to whom she wants. Or she may simply at any point in time refuse to participate further in the protocol. Her goal may be the same as that of the passive adversary: learning something she should not learn. The goal may also be to affect the result of the computation in a way which would not be possible with a TTP. If the adversary succeeds with either goal, we say that the protocol is insecure.

As an example of affecting the output in a way which we do consider a security violation, in Example 3.1 an active adversary may want the output to be exactly

[1]Some information can be deduced from the sum, e.g., no company has input greater than the sum.

\$1,000,000. When a TTP performs the computation, there is no good way for her to achieve this except for by guessing the sum of the other parties' inputs and then selecting her own input appropriately. With an insecure protocol, it may be the case that she can force a specific output, or that she can learn the sum of others' inputs before being forced to select her own input.

Threshold Adversaries

If n parties participate in a computation, we want the protocol to remain secure not only against a single party misbehaving. It may be that several parties in the computations collude to break the security. Thus, we need to consider the ways in which parties may collude.

We model collusions as a single *adversary* who *corrupts* a number of the participants in the protocol. When the adversary corrupts a party, she takes control of the party in the protocol. She learns all the information known to the party in question. Additionally, in the case of an active attacker, she can decide how the party acts in the protocol.

To be able to create secure protocols, we often need to place some limit on the parties an adversary may choose to corrupt. The most common type of restriction studied is a *threshold adversary*. By this we mean that there is some threshold $t \leq n$, and that the adversary can corrupt any set of parties, as long as she corrupts at most t parties. Common values of t are n, $\lfloor (n-1)/2 \rfloor$ (*honest majority*), and $\lfloor (n-1)/3 \rfloor$. These occur as the generic solutions (to securely compute any function, cf. Section 3.3) yield protocols secure with these thresholds in the various models of an adversary's capabilities (e.g., passive or active and computational or information-theoretic security, which we proceed to discuss).

The concept of a threshold is easy to work with, but may not precisely capture what collusions we are trying to protect against. As an example, we may know that two specific parties would never collude. Such knowledge cannot be captured when analyzing a threshold adversary, but may be useful in constructing a more efficient protocol. Alternatively we may be worried about one specific collusion of many parties, but know that all other potential collusions will be small. Such scenarios were treated by Hirt and Maurer [50], who considered an adversary structure, which defines the possible collusions. We use such more general definitions of whom the adversary can corrupt in our Paper IV. There, our protocols are secure as long as an adversary cannot corrupt parties which separate the network on which the protocol is run into two or more components.

A model of an adversary is used when proving a protocol secure. In a real-world execution, however, there is typically no puppet-master who "corrupts" parties, but rather the parties themselves may collude. So how do we interpret a proof of security saying that the protocol is secure when the adversary corrupts at most t parties? Such a proof means that as long as no more than t parties collude, the protocol remains secure. If more than t parties collude, however, we cannot guarantee the security. In a sense, the security is very binary; the protocol offers very strong security guarantees up to a point where security is provided and then no security at all. The topic of making the degradation more graceful has been studied, there is a result along these lines by Lucas, Raub and Maurer [76].

Computational vs. Information-Theoretical

Another distinction made is how much computational time an adversary is willing or able to invest to break the security. Here, two main models exist. Either we consider an adversary with unlimited computational power, or we consider an adversary limited to "practical", but very large, computations (more formally, probabilistic polynomial time). If we achieve security against a computationally unlimited adversary, we say that the protocol is *information-theoretically secure*, and if the protocol is secure against the more limited adversary, we say it is *computationally secure*.

How big is the difference between the two settings? The major difference stems from that fact that we can no longer use traditional ciphers in the case of an unlimited adversary. If such an adversary sees an encrypted message, she could simply try decrypting with all possible keys, until she finds the one that correctly decrypts the message!

One may argue that computational security should suffice for all practical situations. After all, any real-world attacker would have limited (but possibly large) resources at their disposal. So, why work with information-theoretic security? One answer is that such protocols are unconditionally secure: we can mathematically prove their security, and the protocols remain secure independent of other developments. Even if a breakthrough is made in computing technology, for instance if quantum computers become practical, information-theoretically secure protocols remain secure. In contract, quantum computers are known to break some of the computationally secure protocols.

Computationally secure protocols could also be broken by mathematical breakthroughs. All the computationally secure protocols rely on a conjecture known as $\mathsf{P} \neq \mathsf{NP}$, and often more specific conjectures. While many of these are likely to be true, they have not been proven. With a computationally secure protocol, one may also worry that an attacker records all data sent in the protocol, in the hope that she will some day have the resources or algorithms to violate the security and learn information about other parties' inputs. Müller-Quade and Unruh [82] have suggested the concept of long-term security which combines computational and information-theoretical security. There, the attacker is assumed to be computationally bounded while the protocol is executed, but may then use unlimited resources to analyze recorded material from the protocol execution.

3.2 Network and Computation Model

With a few exceptions, most protocols for secure multi-party computation require all parties participating in the computation to be able to speak directly to each other, or in networking terms: a *full mesh* network. In the information-theoretic setting, we furthermore require the pairwise channels between participants to be perfectly private, meaning that they are secure from eavesdropping. This is prohibitively costly to achieve even for moderately large n, as the total number of connection is $\mathcal{O}(n^2)$. Do we really need all these connections? Or alternatively, what can we do if our protocol is to run on a partial-mesh network[2]? When discussing the network topology an MPC protocol is run on, we also refer to the parties as *nodes* in the network.

One approach is to *simulate* a full-mesh network on top of a partial-mesh network. Thus, we construct an overlay network, as in peer-to-peer protocols (cf. Section 2.2). A difference is that require much stronger security guarantees from our overlay here. Solutions to constructing secure full-mesh overlays were discussed by Franklin and Yung [43], and Ashwin Kumar *et al.* [70]. There is, however, a large efficiency penalty in running a protocol designed for a full-mesh network on an overlay over a partial-mesh network. A message that was supposed to be a single message across a single link will instead be forwarded along a number of node-disjoint paths. To successfully be able to simulate a full-mesh network, we require certain properties on the network connectivity. We say that a network is k-connected if there are at least k node-disjoint paths between every pair of nodes. To information-theoretically securely simulate a full-mesh network, we need the network to be $t+1$-connected or $2t+1$-connected (passive [43] and active [70] adversary, respectively) to remain secure against a threshold adversary corrupting up to t nodes.

A second approach is to design secure MPC protocols that can run directly on a partial-mesh network. This means that the protocol can only ask a party to send messages to those which it is connected to. Preferably, such a protocol should be able to run on any network, but sometimes we may need to require some properties of the network. The Dining Cryptographers networks by Chaum [27] can be seen[3] as an early result for summation following this approach. In Paper IV we continue along these lines, giving protocols for additional functions.

Finally, a third approach has also been suggested: to make sure that n is small. How do we do this, if we want to compute functions on inputs contributed from a large number of nodes? We select a subset of parties, called *privacy peers* to perform the computation, and the other parties, called *data collecting parties* contribute their input to the privacy peers in secret shared form, but then do not participate further in the MPC computation. This approach is discussed in e.g. [22, 18].

A different line of research has been to investigate what functions can still be computed in networks with low connectivity, in particular 1-connected networks. A 1-connected network is a network where there at least one pair of nodes such that all their communication must pass through some specific, third, node. Categorizations have been given by Bläser *et al.* [17] and Beimel [13] for which functions can be

[2]Partial-mesh is a network which does not have direct connection between any pair of parties.
[3]It was presented as a protocol for anonymous communication, not summation.

securely computed in 1-connected networks. Such a result of determining the exact limits for what can be achieved typically takes the form of two proofs: a positive result giving a secure protocol for some class of functions, and a mathematical proof that no secure protocol can exist for some other class of functions.

Having discussed the topic of network connectivity, we now return to the classic setting with full-mesh networks. We begin by describing the general positive results, showing that any function can be privately computed. We then move to a discussion on impossibility results, proving that some functions cannot be privately computed beyond some threshold.

3.3 Sharing a Secret and Computing on Secret Shares

Very general solutions for MPC were given by Goldreich, Micali, and Wigderson [46], Ben-Or, Goldwasser, and Wigderson [15], and Chaum, Crépeau, and Damgård [28]. These results showed that any function could be securely evaluated by computing on *secret shared* values, which we discuss next. In our description here, we focus on the latter two results. To describe these, we first explain secret sharing, and then we outline its use in MPC protocols.

Secret Sharing

Imagine you have a very important piece of secret information, that must be kept secret for a long time, and revealed at some date in the future. It is highly important both that the information is kept secret, and that it can be reconstructed later, even if unforeseen events happen. To make sure that the secret is not easily lost, you cannot keep it to yourself (in case anything happens). Thus, you want some mechanism where you also give information to your friends, allowing them to recover the secret. As it is a very important secret, you cannot just give it immediately to them as you cannot fully trust them not to reveal the secret.

The scenario may sound a bit strange, but the problem (and solutions) does arise in practice. A recent example of this is the cryptographic key used to sign the root zone in the DNSSEC protocol. This key is used to protect the lookup from a domain name to an IP address. To protect this key, a number of precautions were taken, as described in [75]. Simplifying slightly, the core of the mechanism is the following: seven trusted officers from around the world were given a piece of information, such that any five of them can reassemble the key.

We refer to the party who begins knowing the secret as the dealer. The problem of a dealer handing out n shares of a secret s such that any t of them can recover the secret (but any choice of $t-1$ cannot) is known as secret sharing. For $t = 1$, the problem is simple as we can just tell everyone the secret. Technically, we require the secret to be in a finite field $\mathbb{Z}_p$, and the arithmetic in the remainder of this section is in this field.

For $t = n$ there is also an easy scheme. Give party P_i a random value r_i for $1 \leq i \leq n-1$. Finally, give the last party P_n the value $s - \sum_{i=1}^{n-1} r_i$. One can simply sum all shares to retrieve the secret s. It can be (easily) proved that learning any choice of $n-1$ shares does not reveal any information at all about s. This type of secret sharing is sometimes used in MPC protocols, for instance in Sharemind [18] and Paper IV.

For more general choices of t, we need something slightly more complicated. Here, we sketch Shamir's secret sharing [98]. The construction is motivated by the fact that a $t-1$-degree polynomial is uniquely determined from t points. The dealer begins by selecting a random polynomial $Q(x)$ of degree $t-1$ with constant term $Q(0) = s$. Each party P_i then receives $Q(i)$, the value of the polynomial evaluated at "their" point.

From t such shares, the polynomial is uniquely determined, and can be computed using polynomial interpolation. From the polynomial, the parties learn its constant term s, which is the shared secret. With $t-1$ or fewer pieces, it can be shown that the parties learn no information at all about the value s.

This scheme works when all parties honestly report their shares when pooling them to recover the secret, as is the case with a passive adversary. But what happens if a party is dishonest and lies about their share, as an active adversary might? Then the wrong polynomial (and thus, secret) is recovered. This problem is addressed by Verifiable Secret Sharing (VSS) [31]. There, even if some of the parties lie about their shares, the correct secret can be recovered. We omit a description of VSS schemes, and simply note that they exist.

Computing on Secret Shares

Having described secret sharing, how do we use it in an MPC protocol? The results of [15, 28] give protocols to add and multiply values under secret sharing. What we mean by this is that if we have the value a and the value b secret shared among the parties, there is a protocol for multiplication that results in the parties also having a secret sharing of $a \cdot b$. For addition, we don't even require a protocol, if each party simply sums her share of a with her share of b, this results in a share of the value $a + b$. This holds for the secret sharing schemes described above, other schemes could require a protocol also for addition.

Such functionality for addition and multiplication is sufficient to compute any function. This can be seen by considering addition and multiplication of Boolean values, which corresponds to the functions XOR and AND. We can represent any function by a Boolean circuit using only these two functions. For most functions, such a representation will be very large. To evaluate a function, the parties secret share their inputs bit by bit, and the function is then evaluated gate by gate with multiplications and additions. After all gates have been evaluated, the parties have the output values in secret shared form. They then pool the shares of these values, revealing the output of the function.

Depending on what level of security is targeted, the protocol may run over a standard secret sharing scheme such as [98] or a VSS scheme such as [31]. The difference between these two approaches is whether the result protocol is secure against a passive or active adversary. In the information-theoretic setting, the general protocols are secure with threshold $\lfloor (n-1)/2 \rfloor$ in the passive case, and with threshold $\lfloor (n-1)/3 \rfloor$ in the active case. These are also the thresholds used in the secret sharing schemes.

We remark that there are other general solutions to MPC. A significantly different approach is based on the idea of a *garbled circuit*, and was presented by Yao [108] for two-party computation (i.e., when $n = 2$). Garbled circuits are used between pairs of parties in [46], and were generalized to the multi-party case by Beaver, Micali, and Rogaway [12]. We omit further discussion on the garbled circuit paradigm.

Having touched upon the positive results that we can compute any function securely, we may ask what the limits of secure computation are. For instance, is threshold $\lfloor (n-1)/2 \rfloor$ the best we can do in the information-theoretic passive case, or could we hope for a better general protocol? Thus, we turn to the subject of impossibility results.

3.4 Impossibility Results

Apart from coming up with protocols to securely compute a function, it is also important to understand the limits of what can be done. An impossibility result is a mathematical proof, proving that something is impossible to do. Two reasons as to why it is important to prove impossibility results are that it saves research resources, and it helps in understanding the problem at a more fundamental level, which may lead to improved protocols. An impossibility result may show that one approach will never succeed, diverting research effort into alternate approaches.

In the area of MPC, we are interested in what threshold is the best possible for some specific function f, or class of functions. As one may imagine, this value depends on the adversarial model used. Here, we consider only information-theoretically secure protocols with a passive adversary.

We say that a function is k-private if there exists a protocol to securely compute it in this setting with threshold k. The general results [15, 28] then show that all functions are $\lfloor (n-1)/2 \rfloor$-private. But is that the whole story? No, for instance the summation function we used as Example 3.1 is n-private[4]. On the other hand, for the Boolean OR function, it was shown [15, 71] that $\lfloor (n-1)/2 \rfloor$-privacy is the best possible.

The general problem of finding the threshold for any function f is a long-standing open problem. There exist a number of partial results. A first question one may ask is if the hierarchy is complete, i.e., is there for all t some function which is t-private but not $t+1$-private? As we know that all functions are $\lfloor (n-1)/2 \rfloor$-private, there cannot be any functions below that level. What about the other levels? Chor, Geréb-Graus, and Kushilevitz [30] proved that in general the hierar-

[4]Actually, n-privacy is equivalent to $n-1$-privacy; if the adversary corrupts all parties, she already has complete information and cannot learn anything more from the protocol execution.

chy is complete: for every $\lceil n/2 \rceil \leq t \leq n-2$ there is a function which is t-private but not $t+1$-private.

One way of restricting the problem is to consider the case of few parties. In particular, the case when $n = 2$, known as two-party computation. In this case, the general MPC solutions cannot be used, as they give 0-privacy in this case, i.e., the protocol is only secure as long as neither party attempts to break the security. Beaver [11] and Kushilevitz [71] independently gave a complete characterization of the functions that can be computed 1-privately in the two-party case. They gave a 1-private protocol for functions which are *decomposable*, and also proved that a function which is not decomposable cannot be 1-privately computed. We use a generalization of the notion of decomposition in our proofs in Paper II.

Another restricted formulation is to analyze functions with a fixed number of distinct outputs. This approach was taken by Chor and Kushilevitz [33] who gave a complete characterization of the privacy threshold of all Boolean functions, i.e., functions with a single bit as output. What they showed was a so called zero-one law: a Boolean function is either $\lfloor (n-1)/2 \rfloor$-private (the lowest possible), or n-private (the highest possible). In Paper II, we extended their result to functions with a single trit (value 0, 1, or 2) as output, and showed that a zero-one law also holds for such functions.

We now go into a bit more mathematical detail on the proof of the zero-one law for Boolean privacy, and related techniques. We use these in the proof of our Paper II.

Embedded ORs and Partition Proofs

A core idea in the proof of Chor and Kushilevitz is to make use of the fact that OR cannot be 1-privately computed. But OR is a function with two inputs, how can we make use of this in the n-party case? We consider functions f that contain an OR as a subfunction, which we call an *embedded* OR. We give a definition from [63] of an embedded OR, but we remark that we use a different (more general) definition in Paper II.

Definition 3.1 (Embeded OR) *A 2-argument function f contains an* embedded OR *if there exists inputs x_1, x_2, y_1, y_2 such that $f(x_1, y_1) = f(x_2, y_1) = f(x_1, y_2) \neq f(x_2, y_2)$.*

An n-argument function f contains an embedded OR *if there exists indices i, j and values a_k such that the 2-argument function*

$$h(x, y) = f(a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_{j-1}, y, a_{j+1}, \ldots, a_n)$$

contains an embedded OR.

We claim that a protocol for a function f which is $\lceil n/2 \rceil$-private would yield a 1-private protocol for OR. If there was such a protocol for f, two parties wanting to compute OR could use the protocol, having one party *simulate* $\lceil n/2 \rceil$ parties, and having the other simulate the remaining $\lfloor n/2 \rfloor$ parties. They would select the inputs simulated for these parties such that the protocol computes OR. By the privacy guarantees of the assumed protocol, this would be 1-private, as each party controls no more than $\lceil n/2 \rceil$ parties, and the original protocol was $\lceil n/2 \rceil$-private. As we know that no 1-private protocol for OR exists, we conclude that no $\lceil n/2 \rceil$-private protocol for a function with an embedded OR can exist.

Having proven this, what remains to get the result of [33] is to give a classification of all Boolean functions. As it turns out, a Boolean function either contains an embedded OR, or it can be written as a (Boolean) summation, $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} f_i(x_i)$. Sums can be computed n-privately, and thus the zero-one law for Boolean functions is proved. In the case of functions with a trit as output one can prove an analogous, but more complex, structure lemma.

The technique of proving that functions cannot be $\lceil n/2 \rceil$-privately computed by showing that they contain an embedded OR has been used in many results since. The technique was generalized by Chor and Ishai [32] to consider partitions into more sets, which was showed to yield stronger impossibility results. In Paper II we only need partitions into two sets.

This concludes our summary of the background in MPC. We proceed with a discussion on our contributions, and the real-world applications of MPC.

3.5 Contributions

In this thesis, we include three papers on the topic of secure multi-party computation. Two of these are focused on the more practical side of the area: constructing secure protocols for important functions. The third paper explores the limitations by investigating what functions can still be securely computed even if half the participants or more collude to break the protocol.

In Paper IV we propose protocols for some basic functions which do not require private channels between every pair of participants. Instead, our protocols can run on any network. This opens up some new applications, an example of which is joint network monitoring by a number of competing Internet Service Providers. We provide protocols for three basic functions: summation, disjunction (OR), and computing the maximum input. As we discussed in Section 3.3, summation and disjunction is sufficient to evaluate any function. This is not the case here, however, as our protocols inherently reveal the output to the parties, while the protocols discussed in Section 3.3 take input and produce output in secret shared form.

Another contribution is in Paper III where we construct efficient protocols for sorting, and a related problem of aggregating data sets. Our protocols here make use of results on *sorting networks* to sort data using few comparisons. We build on the general MPC protocols and our protocols were implemented and tested on the Sharemind platform to measure their performance. The evaluation showed that lists of up to 16000 entries could be sorted within a few minutes. These protocols could be used to build a joint Intrusion Detection System across multiple organizations.

Finally, Paper II extends the zero-one law for Boolean functions by Chor and Kushilevitz [33] to functions with a trit as output. We show that, surprisingly, an analogous zero-one law to the Boolean case holds. By this, we mean that for each function f with a single trit as output, it is either the case that it is n-private, or it is $\lfloor (n-1)/2 \rfloor$-private and not $\lceil n/2 \rfloor$-private.

3.6 Real-world Applications

Despite the fact that MPC protocols to securely compute any function were published more than 20 years ago [15, 28], their real-world use has so far been limited. As the security offered by MPC protocols is very high, and they can perform many general tasks, one may ask why they are not in more widespread use. It is likely that a number of factors contribute to this, including technical factors such as issues with scalability, and performance issues in naive implementations based on the general protocols. In particular, many protocols do not scale well in n, the number of parties participating in the computation. Furthermore, the requirement that all parties can securely communicate pairwise is also problematic in some scenarios.

There are also non-technical obstacles preventing widespread use of MPC techniques. Firstly, the results in the area and their implications are not well-known outside the cryptographic research community. Secondly, several of the traditional roles as TTP are lucrative to the TTP, giving them economic incentive to maintain status quo. Additionally, in some scenarios, the requirement to use a TTP may be mandated by law.

Auctioning Sugar Beet Production Rights

One notable real-world use of MPC was by Bogetoft *et al.* [19] who executed a secure auction. An auction is a real-world example which is normally solved with the assistance of a TTP, in this case an auctioneer. Their application was to reallocate production rights for sugar beets. A production right in this context is a contract allowing a farmer to produce and sell a specific amount of sugar beets per year to Danisco, the only sugar-producing company in Denmark.

The production rights can be sold between farmers, something which has occurred to a limited extent historically. Due to drastic reductions in support for sugar beet production by the European Union, a more complete reallocation of production rights were needed than what can easily be accomplished by bilateral agreements between individual farmers. Thus, it was decided to run an auction in which a total of 1200 bidders participated.

Why could the auction not be run using a TTP? As the bid submitted by a farmer reveals information on how profitable their business is, Danisco could not serve as auctioneer due to the concern that they may use the information learned in future negotiations with farmers. The contracts in some cases act as security for debts farmers have to Danisco. This meant that Danisco could not accept a solution where the farmers' association would serve as TTP. Recruiting some outside TTP was considered too expensive.

The computation was done using a privacy peer construction. Each farmer used their own computer and software provided by the team of researchers. The software then sent the farmer's inputs in secret shared form to three computers, one run by Danisco, one by the farmers' association, and one by the research team. These three computers ran an MPC protocol to compute the result of the auction.

Developer Tools

Apart from the sugar beet auction, there have been other developments pointing in the direction that MPC protocols may see more real-world applications in the near future. One of these is that a number of software projects have been developed that allows programmers to more easily implement MPC programs. These frameworks consist either of libraries implementing MPC primitives in some standard programming language such as Java or Python, or contain their own programming language in which to write MPC programs. With this support, the level of expertise required for programmers to make use of MPC techniques is significantly reduced.

One framework is Fairplay [78] which is only for the two-party ($n = 2$) case and its extension to the multi-party case, FairplayMP [14]. Another is Sharemind [18] which is specialized for the case when $n = 3$. Finally, two other, general, frameworks are SEPIA [22] and VIFF [38].

Applicability of Contributions of this Thesis

We believe that the contributions given in Paper IV and Paper III can be of practical application. The protocols for partial-mesh networks presented in Paper IV could be applied in settings such as network monitoring, joint intrusion detection systems, and also in the context of sensor networks.

Secure protocols for sorting also have a number of applications. In addition to potential application in joint intrusion detection, we believe that they could be applied in database settings. Companies or government agencies could use the protocols to find entries in common in databases, without revealing any further information apart from the entries in common.

3.7 Conclusions and Future Work

The area of Secure Multi-Party Computation studies a fundamental problem in how to cooperate without trust. There exist very general solutions, and the security properties provided by MPC protocols are very good. Yet there have been relatively few applications of MPC in the real world, or even broader areas of science beyond the field of cryptography where the design and properties of the protocols are studied. Thus, one line of future work includes raising awareness of MPC protocols, making them more practical, and applying them as parts of more complex and complete security mechanisms.

One venue of research is to more fully investigate MPC protocols which can run on partial-mesh networks. This includes both investigating positive results, creating protocols which adapt to the network upon which they are run, as well as further investigating impossibility results in more general classes of networks.

Another area is the continued investigation of the thresholds at which various functions can be computed. While there are partial results, the general problem is still far from solved.

Chapter 4

Web Security

One interesting case of security architecture is the World Wide Web, and in particular the security implemented by a web browser. Many financial transactions are done over the web, including bank transfers and credit card payments. Apart from the integrity of financial transactions, there are also privacy issues. We access private information and communicate via web pages and do not want a malicious server to be able to spy on our activity.

General security principles and problems apply to the web, and attacks and defenses in web applications and browsers are often well studied and have specific names. As an example, the *cross-site request forgery* (CSRF) attack which we discuss in Section 4.2 is an instance of the *confused deputy* problem. Here, we focus on the web-specific attacks and use the web-specific terminology.

The Open Web Application Security Project (OWASP) publishes a list [87] with the top ten web related security issues. These include both attacks which are performed by an attacker directly against a server, such as injection attacks, but also attacks which affect a web browser or the interaction between browser and server, such as CSRF attacks. Here, we focus on the latter class and describe in some detail three attacks: cross-site request forgery, cross-site scripting, and history detection. Our contribution in web security, Paper V, builds on history detection and has some similarities to a cross-site request forgery attack. Cross-site scripting is the largest class of web vulnerabilities, and provides background for a discussion on JavaScript security policies. Sometimes, several minor vulnerabilities can be combined to increase their impact. After having described basic vulnerabilities, we include a discussion on this with two examples, including our Paper V. We conclude this chapter with a summary of the contributions of this thesis.

To begin discussing security issues pertinent to the web, we must first understand how web sites keep track of their users. When a web browser visits a page, it makes a connection to the server hosting the web page. If the user then follows a link on the page, the web browser makes a new connection to the server and requests the new page indicated by the link. By default, the server has no information on who it is that makes a request, or even that two requests are made by the same user.

4.1 Keeping Track of Users

Many web sites have a way for users to log in, so clearly there has to be some way for a server to know which user is making a request. This is done through a mechanism known as cookies. A cookie is a small piece of information that a server can give to a web browser, and which the web browser sends back to the server in each request. This seems to offer a very easy solution: once a user has successfully logged in, the server could set a cookie with the username to remember who the user is. When Alice logs in, the server could set the cookie `user=alice`. Her browser would include this cookie in all future requests, so the server would know that those came from Alice.

We think of an attacker, Mallory, who would like to access Alice's account on the service. One way for Mallory to access a cookie is if the cookie is sent unencrypted over a network she is connected to. When using the encrypted HTTPS protocol, the cookies are protected, but many sites still use plain HTTP, where the traffic is not encrypted. To illustrate the risk of cookies being stolen over the network, an application called Firesheep was developed by Butler [23]. It received much publicity after its release in late 2010, prompting large sites like Facebook and Twitter to migrate to HTTPS. Firesheep listens for unencrypted cookies on a local network (e.g., a wireless network at a coffee shop or conference) for popular services like Facebook and Gmail. It provides a convenient GUI to show whose cookies it has captured, and to log in using a stolen cookie.

We assume that the cookies we consider here are only sent over HTTPS. With our current proposal, Mallory could easily log in as Alice, if she only knows Alice's username. It is reasonable to assume that Mallory does know Alice's username, but not her password. Mallory could then simply set the cookie `user=alice` in her own browser and connect to the server to be logged in as Alice.

The problem with the username in the cookie was that it is very easy for an attacker to *guess* a cookie value which would log her in. This could be solved by a cryptographic construction known as a Message Authentication Code (MAC), but the most common mechanism is to use a *session cookie*. A session cookie is a cookie where the information is simply a long random number, so long that it would take too long for an attacker to guess a number which is in use. When a user visits the web site, the server gives them a session cookie, consisting of a random number, for instance `session=1445975142` (in reality, the number is larger). In every subsequent request they make to the server, the same number will be present. This allows the server to know that a series of request is made by the same browser.

When a user logs in, the server itself stores some information associating the number in the session cookie with the username of the logged in user. Thus, on the server, the mapping that session 1445975142 corresponds to Alice is stored once

Alice has logged in. If Alice logs out, the mapping is removed, and the session cookie becomes useless. To become logged in as Alice, Mallory would have to somehow learn the session number of Alice. If she learned this number, then she could set the cookie `session=1445975142` in her own browser and access the site, being logged in as Alice.

The use of cookies to handle user logins means that it becomes important for a browser to protect its cookies. For instance, if a user visits `evil.com`, the server for that site should not be able to access the user's session cookie for her bank. As a first measure of protection, a cookie is tied to a specific domain name, and is only sent back to servers in the same domain as where it was set. Other mechanisms are also in place, in particular restrictions on how JavaScript programs may interact between pages served from other domains, but we defer that discussion to Section 4.3. The key point is that if an attacker can somehow get hold of your session cookie for a site, she is logged in as you and can do all actions you can when logged in[1].

From the discussion on a session cookie, we saw that it is important to protect the cookie. Can Mallory still attack Alice in some other way than by stealing her cookie? Yes, there is more to the story, and we now discuss three other classes of problems in web security.

4.2 Cross-Site Request Forgery

Recall that the session cookie is included in every request the user makes to the server. Some requests not only fetch a web page for the user to view, but also have *side effects*, such as placing an order or making a bank transfer[2]. To identify who is making a bank transfer, the session cookie included in the request is used. When Alice wants to make a bank transfer of $100 to Bob, she would go to her bank's web site and enter the details of the transaction. Finally, when she clicks the confirmation button, her browser goes to a link looking something like `bank.com/transfer?amount=100&to=Bob`, and the browser would include Alice's session cookie, identifying her to the bank.

What happens if Mallory can trick Alice into following a link to `bank.com/transfer?amount=100&to=Mallory`? Alice's browser would include Alice's session cookie with the request, so the bank would transfer money from Alice to the attacker. If Alice is very observant, she may notice that the link looks suspicious and refuse to click on it. Even then, Alice would still be susceptible to the following attack. A web page can include parts which come from different servers, the so-called *iframes* are one example of this. So Mallory may may set up `evil.com`, a page which would include an iframe with address `bank.com/transfer?amount=`

[1]This is not true in all systems; additional security measures such as tying a cookie to the IP address from which the user connects are sometimes used.

[2]Here we consider a simple bank web page. Some banks have additional protection mechanisms where a user must sign off on transfers by means of a small hardware device or a one-time code.

`100&to=Mallory`. If Alice were to visit `evil.com`, her browser would automatically also make a request to transfer money, unbeknownst to her.

This type of attack, where one site causes a visitor's browser to make a request with side effects to a second site, is referred to as a *Cross-Site Request Forgery* (CSRF). This is an attack where viewing one page automatically causes a side effect on some other site. The problem here is essentially that visiting a confirmation URL *should* perform a bank transfer when the user makes a bank transfer, but visiting the same address with the same session cookie should not if the request was made automatically without the user's knowledge. How is the bank's server to distinguish between the two cases?

What distinguishes a legitimate request from a request caused by a CSRF attack? A distinction is that a legitimate request occurs because the user fills out a form or follows a link on the bank's web site, while the CSRF attack causes the request to come from a third-party page. When designing a defense mechanism, we should try to make use of this difference.

A number of mechanisms have been proposed [9, 99], we only describe one here: a *CSRF token*. This is entirely a server-side protection mechanism. A CSRF token is another random value, specific to the session, but separate from the value of the session cookie. This value is inserted by the bank into the URLs of internal links. If Alice's CSRF token is 7401932, the link for her to set up a transfer would be `bank.com/transfer?amount=100&to=Bob&token=7401932`. The server when receiving the request verifies that the token is present and matches Alice's session. If the token is missing or wrong, the server does not perform the transaction but instead displays an error message. As the CSRF token is random and specific to Alice's session, our attacker Mallory cannot guess it, and thus does not know the right link to direct Alice's browser to.

We now turn to the next attack, cross-site scripting. This is a vulnerability which can be used to steal session cookies as well as circumventing CSRF defenses on vulnerable pages.

4.3 Cross-Site Scripting

One of the most frequent security problems in web security is known as *cross-site scripting* (XSS). It was the most common vulnerability reported in MITRE's database of Common Vulnerabilities and Exposures (CVE) in their 2007 statistics [34], and it is item number two on OWASP's 2010 top-10 list [87]. This type of vulnerability has been publicly documented for over 10 years [26]. We now explain what an XSS attack is, and hint at why it is so prevalent.

Before this, we need to cover a bit of background on scripting on the web. Web pages may contain programs, or scripts, written in a language called JavaScript. These programs can read and modify the content of the page that is displayed. As scripts can be automatically executed as soon as the browser visits a web site, there clearly needs to be some restrictions placed on what they can do (for instance, we do not want a malicious web site we visit to be able to read information from a bank site we have open in another tab). JavaScript can be included anywhere on a web page using the HTML tag `<script>`.

When adding security restrictions in the JavaScript language, it is important to not hinder useful, legitimate functionality more than necessary. Browsers need a good *security policy* that prevents attacks, allows legitimate functionality, and that can be understood by developers. The common solution, first implemented in Netscape Navigator 2.0 in 1996, is the *same-origin policy*. This policy states that scripts on a page cannot access information on pages from a different origin. Origin in this context can be thought of as meaning server[3]. This means that the security permissions of a script depends on which web site includes the script's code.

If an attacker could trick a bank's web server into including a malicious script when a victim visits the bank's web site, then the security context of the script is that of the bank's web server. This means that a malicious script could steal the victim's login credentials, session cookie, or CSRF token. Additionally, it could access the victim's bank information or perform transactions. In XSS attacks, the attacker is attacking a session between two other parties, here the bank and the user. We refer to the server as the *target*, and the user as the *victim*.

Reflected and Persistent XSS

How could an attacker get a bank to include her malicious JavaScript on their web page? There are two primary types of XSS attacks, *reflected* and *persistent*. A third type, called DOMXSS has also been published [64], but we focus here on the two main types.

We begin with the case of a reflected XSS attack, and begin introducing it by means of an example. Consider a search box on the target's web site. When searching, most web sites include the sought for terms in the resulting web page, e.g. "Your search for life gave 3,770,000,000 results". This may be sent in HTML as `<p>Your search for life gave 3,770,000,000 results</p>`.

What happens if we search for ``<script>alert(``Hello'');</script>``[4]? A vulnerable web site would return the text in the resulting HTML, ``<p>Your search for <script>alert(``Hello'');</script> gave 45,700,000 results</p>``. This is legitimate HTML, with a script to be executed. Thus, the browser would execute the JavaScript, in this case simply displaying a dialog window saying "Hello." The issue here is that the web server did not sanitize the input before including it as part of the output.

The argument to search for is typically part of a URL, so having found a vulnerable page on the target server, the attacker could construct a link that corresponds to a "search" for a script she wants the victim to execute. She would then trick the

[3]The same-origin policy restrictions apply if there is a difference in protocol, host name, or port. For a more full description, we refer to Zalewski [109].

[4]Alert is a JavaScript function which opens a dialog. It is commonly used as a demonstrator of XSS vulnerabilities.

victim into following the link. As the script is sent to the victim's browser from the target web server, it has the security permissions of any script served from that site. What we just showed is known as a reflected XSS attack; the link contains malicious data which the target server reflects back to the victim's web browser.

In the case of a persistent XSS attack, the attacker instead looks for ways of including information directly on the server. In the case of a bank, it is often the case that bank transfers may include short messages to the recipient. These messages are displayed to the recipient when they look at their bank statement in the bank's web interface. The attacker may thus send the victim \$0.01 with a message of `<script>alert(``Hello'');</script>`. If the bank's site is vulnerable, this JavaScript would be passed on to the victim's browser every time the victim looks at her bank statement. This is a persistent XSS; the attacker found a mechanism to post data to the target server that is included in a web page.

These two example attack vectors may help in explaining why XSS attacks are so prevalent. An XSS attack vector exists any time user-controlled information is sent back on a web site without proper sanitization. Typical web sites have a huge number of input paths, and failing to sanitize a single path from input to inclusion in output results in an XSS vulnerability.

We now proceed to the third and last attack we discuss, history detection.

4.4 History Detection Attacks

Another long-standing security problem in the area of web security is is history detection. Here, we primarily discuss a *single*, specific problem that has been publicly known since at least 2000 [96]. This vulnerability is due to unforeseen interaction between to legitimate, standardized functionalities. This means that all browsers have traditionally been vulnerable to the attack as they have all implemented the standardized functionality. A defense has now also been standardized, and implemented in most of the major browsers.

The attack uses the fact that a visited link is styled differently from an unvisited link, and is referred to as *CSS history detection*. The name comes from Cascading Style Sheets (CSS), used to describe layout of web pages. We remark that there are other attacks to achieve history detection, an attacker may also measure how quickly elements load to determine if they were cached or not as outlined by Felten and Schneider [42], but in this discussion we only detail the CSS history detection flaw.

In this attack, the attacker who operates `evil.com` wants to learn if her visitors have also visited `example.com`. The basic flaw, as discussed in Section 1.2 is that the style applied to a link on a page can be read by JavaScript executing on a page. Since visited links are styled differently from unvisited links, `evil.com` can include a link to `example.com`. JavaScript on `evil.com` can then detect if the link is styled as if it is visited or unvisited, revealing whether the visitor has visited `example.com` or not. We remark that the same-origin policy discussed in Section 4.3 does not prevent this; the script is running on `evil.com`, and the link element is likewise on the `evil.com` page, so the script is allowed to access information about the link.

A proposed fix to the CSS history detection hole by Jackson *et al.* [52] was to essentially apply the same-origin policy for the visited history of a browser, something they also implemented in the form of the SafeHistory extension. Their suggestion was not adopted in any of the major browsers, however. Instead, a different defense mechanism was later suggested by Baron [8], and has now[5] been implemented in almost all major browsers. This defense mechanism severely restricts how the visited attribute of a link may affect styling. In essence, only the color of a link may be affected by if it is visited or not, and a JavaScript querying the color of the link will always get a return value as if the link was unvisited.

Several uses has been proposed for history detection, both maliciously and helpful to the user. Using history detection to assist a user, a web site may detect if the user has visited known malicious sites and warn them that they may have been infected with malware [53]. Janc and Olejnik [54] investigated the privacy implications of the attack and reported that they were able to detect probable US ZIP code for 9.2% of tested users. Their study was done before the major browsers had implemented defenses.

Having finished our discussion on the specific attacks we cover, we continue with a discussion on how an attacker can combine multiple attacks.

4.5 Combining Attacks

In security in general, an attacker may sometimes chain together attacks to achieve their goals. A good example of this in web security is a combination of XSS and CSRF attacks. Recall that the primary mechanism to defend against CSRF attacks is the inclusion of CSRF tokens on the target web site. Normally, the same-origin policy prevents an attacker's page from accessing the CSRF token on the target site. If the target has an XSS vulnerability, however, injected JavaScript can read the CSRF token, which allows the attacker to then perform CSRF attacks.

For a more complex example, we turn to a presentation by Kamkar titled "How I Met Your Girlfriend" [60]. Here, the attacker wants to know where Bob's girlfriend Alice lives. The chained attack begins by exploiting a vulnerability in the randomness in the session cookie generated by the PHP programming language [61]. As we discussed in Section 4.1, the session cookie should be a long random number. As it turned out, the numbers used by earlier versions of PHP were not sufficiently random, and could be guessed by an attacker. Using this, the attacker guesses Bob's session cookie and is thus logged into his account on a social networking site. The attacker then posts a message to Alice, tricking her into visiting `evil.com`, a domain under the attacker's control.

On the `evil.com` domain, the attacker uses a reflected XSS attack against Alice's home router. Such devices almost always have a built-in web server to

[5]Firefox 4 and Internet Explorer 9, both implementing this were both released March 2011.

configure them using a web interface. The reflected XSS reads out the so-called MAC address of Alice's wireless network. The MAC address is a globally unique number identifying the wireless network interface in Alice's router. A MAC address is no big secret; it is broadcast over radio in the local area by the wireless router, so they can be easily collected by someone nearby. The vulnerability here is that our attacker was able to retrieve the address across the Internet through an XSS hole in the victim's home router.

How do we go from the MAC address to a geographic position? As it turns out, Google has a database that allows a lookup of the approximate position of a wireless router from a MAC address. The reader may at this point wonder why such a database exists. This service is used for instance by the Firefox browser's implementation of geolocation services [81]; most computers do not know where they are located, so for Firefox to provide location services, the computer looks for nearby wireless routers, collects their MAC addresses, and sends them to Google to learn its approximate position.

The latter part of the attack is a good illustration of how a seemingly harmless attack (learning the MAC address of a home router) combined with legitimate functionality can have unforeseen consequences. In this case, a malicious web site visited by the victim could learn her home address.

4.6 Contributions

Included in this thesis is Paper V on web security, detailing a new attack we call *Flow Stealing*. The attack builds on the fact that a web page retains control over windows which it has opened. In particular, on most browsers, a web site can at any point in time navigate a window it has opened to a new address. Why would that be a security problem?

The core scenario in the attack is as follows: our attacker runs a malicious web site, visited by the victim in tab t_1. The web site has a link to a store, opening in a new tab t_2 when clicked. The victim browses the store, selects goods to buy, and proceeds to pay. Payments are handled by a payment provider, external to the store, so when the victim is about to pay, the store (legitimately) redirects her browser to the payment provider's web site. The attacker's site which is still open in tab t_1 in the victim's browser detects that the victim hits the payment provider. At this moment, the attacker's JavaScript in tab t_1 redirects the tab t_2 to the same payment provider, but with the attacker as recipient of the payment.

To detect when the victim reaches the payment provider, the attacker uses the CSS history detection attack in a new way. She repeatedly tests if the victim has visited the payment provider. We assume that the payment provider begins as unvisited in the victim's browser. When the visited status of the payment provider's page switches from unvisited to visited, the JavaScript on the malicious web site triggers the redirect.

Chapter 5

Broadcast Encryption

Broadcast encryption is the problem of how to distribute encrypted content to a large, dynamic group of recipients from a single, trusted sender. Typical application include a streaming system or traditional cable TV. These two use cases both concern protecting a media stream in an online setting. Another application is to protect media content offline, by using broadcast encryption to distribute content-specific keys on discs like DVDs or BluRay discs. In fact, broadcast encryption is used today for content protection on BluRay discs. We return to this in Section 5.5.

While the typical motivation and applications all involve content protection, there are also other uses for broadcast encryption schemes. Another application is to enable secure group communication, where we want to establish a shared secret known only by all current members of a group and where the membership changes over time.

As we discussed in Section 1.2, the core problem in broadcast encryption is to establish a joint secret that is known to all current members, but not known to any non-members. We recall that a mechanism solving the problem is known as a *broadcast encryption scheme.* By focusing on constructing a mechanism to establish and maintain such a joint secret, we can mostly ignore if we are solving a problem in streaming, offline content distribution, or group communication. We want our broadcast encryption scheme to be as efficient way possible, where our measure of efficiency is the bandwidth used by the sender, which we want to be as small as possible. Other important performance parameters, that we for the most part ignore, are the amounts of memory and computation time needed by both sender and receivers.

We proceed to define the problem and discuss some of the proposed schemes from the literature. We go into more detail on subset cover schemes in Section 5.2, as our work in broadcast encryption relates to such schemes. After this, we touch upon the subject of relaxing the security requirements in broadcast encryption to gain efficiency. Finally, we summarize the contributions of this thesis and discuss the key real-world application of broadcast encryption.

5.1 Defining the Problem

We begin by establishing some terminology for the recipients in a system. A broadcast encryption is thought of as having a universe of *users* comprised of everyone who may ever receive content from the system. At any given moment, a user may be a *member*, or may be *revoked*. We denote the number of members by m, and the number of revoked users by r. We denote the total number of users by n, which is equal to $r + m$. The system's purpose is to establish and maintain a shared secret which is known only by all current members and the sender. We recall that this shared secret is called a *media key*.

Many schemes require the number of users to be constant and fixed before the system begins broadcasting. This is a problem in some applications, e.g. streaming. There it is difficult to know how many different users will use the system, and impossible to identify all potential customers for the lifespan of the system in advance! One solution is to set the system up with a very high number of users n. All "users" would be initially unassigned, and depending on the system, unassigned users may either be treated as members or revoked. When a new user becomes a member for the first time, she is then assigned the identity of the first unassigned "user" in the system.

In the standard definition of a broadcast encryption scheme, there are two steps. In the first step, initialization, keys are generated for all users of the system and for the sender. Each user u_i then receives her unique set of secrets s_i. These secrets allow her to efficiently compute the cryptographic keys she should have. In a few systems, the set of secrets is simply a number of keys. In most system, however, a user has access to so many keys that it would be cumbersome to store them all. We want users to only have to store a "reasonable"[1] amount of secrets. The initial distribution of secrets to a new user is assumed to happen in some secure way.

The sender also needs to know the keys that the users have, so it can encrypt with them. To simplify our discussion, we assume that the sender has access to all keys. In a real system it is important, just like for the users, to also ensure that the sender is able to compute all keys from some set of secrets of reasonable size.

Having been initialized, the system repeatedly is asked to send out a new media key to the current set of members. It may be that this is done every time a user joins or leaves, or it may occur periodically. There should not be a limit on how many times the system can distribute a new media key[2]. When the system distributes a new key, the sender sends out a message to all users. The message has the property that a user u_i who is a member can use her secrets s_i together with the message to recover the new media key, and that a revoked user u_j cannot recover the media key using her secrets s_j. The size of such a message is the *bandwidth overhead* of the system.

[1]The systems we discuss require users to store $\mathcal{O}(\log m)$, $\mathcal{O}(\log n)$, $\mathcal{O}\left(\log^2 n\right)$, or $\mathcal{O}\left(\sqrt{n}\right)$ elements.

[2]The earliest broadcast encryption schemes, e.g. one by Berkovits [16], could only distribute a limited number of keys.

We now discuss the security we want a broadcast encryption system to have, and then turn to the main solution strategies to construct broadcast encryption schemes. Finally, we discuss the related concept of identifying dishonest members, known as traitor tracing.

Security of Broadcast Encryption

How do we define the security of a broadcast encryption system? We want to protect against revoked users being able to access the media key. It is conceivable that multiple revoked users would collude in order to access the media key, so we would prefer the system to provide security against such attacks. To capture collusions we think of a single adversary attacking the system, who corrupts users as in our discussion on secure multi-party computation in Section 3.1[3]. When the adversary corrupts a user u_i she is given the user's secrets s_i. The adversary's goal is to learn the media key. In the case of broadcast encryption, we only consider the case of computational security, i.e., we assume the attacker has limited (but very high, probabilistic polynomial time) computational power.

Could we tolerate an adversary who corrupts all the users in the system, or do we need to restrict her powers? Yes, we do need to restrict our adversary. If she has corrupted a member, then she can just follow the normal decryption process using the secrets of the corrupted member to recover the media key. This cannot be prevented, as the mechanism should deliver the media key to all members. Thus, we see that the adversary must be restricted such that she can not corrupt any members.

As it turns out, we can build secure schemes with only this restriction; we can tolerate an adversary corrupting all revoked users. This corresponds to a very strong security property when a broadcast encryption system is used: even if all revoked users were to collude to break the system's security, they would not succeed. On the other hand, if a single member is dishonest, she can re-distribute the media key to revoked users every time it updates, allowing them to also view the content. The problem of dishonest members is mitigated by what is known as *traitor tracing*, which we return to later in this section.

Solution Strategies

The problem of broadcast encryption was introduced by Berkovits [16], who suggested a solution based on polynomial interpolation. The idea builds on Shamir's secret sharing [98] which we previously discussed in the context of MPC in Section 3.3.

Since the introduction of the problem, and the first scheme, many new schemes have been proposed. We recall from Section 1.2 that an important property of a scheme is whether it is *stateful* or *stateless*. In stateless systems, users' secrets s_i

[3]Such a model is used in most of cryptography, not only the two areas we discuss in this thesis.

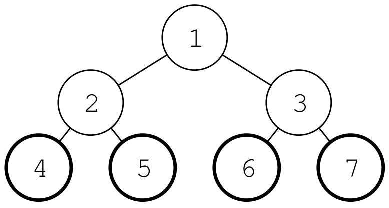


Figure 5.1: The keys in a LKH system with four users (at positions 4–7).

are never updated, while in a stateful system, the secrets of users may be updated by a message delivering a new media key. This means that a stateful scheme can be difficult to use in a setting where users may not receive all messages, and must often be augmented with a resending mechanism. We now discuss some stateful and some stateless schemes.

Stateful

An early broadcast encryption schemes is the Logical Key Hierarchy (LKH) scheme, proposed by Wallner, Harder and Agee [105] and Wong, Gouda and Lam [106] independently. In this scheme, a binary tree of keys is maintained. Each leaf node corresponds to a member, and the internal nodes of the tree are only used for key management. A member knows all the keys on the path from its leaf to the root node. As long as the tree is balanced, this means that each member will need to store $\mathcal{O}(\log m)$ keys, this is her set of secrets s_i. Revoked users do not have any keys in the tree, but do share a key with the sender so the sender can send them new keys if they join. We illustrate an instance of the LKH system with four users in Figure 5.1. In this example, there are four users, at positions 4–7 in the tree. The user at position 6 knows the keys associated with nodes 6, 3, and 1.

Assuming we have a tree with these properties, how can we handle updates in the membership? We proceed to describe algorithms to handle a single user joining, and a single member leaving. To handle larger updates, these operations can be repeated. We remark that it is also possible to more efficiently handle a number of membership changes simultaneously.

When a user joins, she needs to be inserted as a leaf to the tree. This involves adding a new internal node in the tree where there was previously a leaf. The member associated with that leaf needs to receive the key for the new internal node. Furthermore, the newly joined member is sent all the keys on the path from her leaf to the root. These are sent to the newly joined member using the secret key shared between her and the sender. This is slightly simplified; we also need to replace all keys that the newly joined member receives. This is to prevent her from being able to decrypt content sent before she joined. As these keys are not known to any attacker, we can send out the updated keys encrypted under the old keys, as everyone who had the old key should also have the new key. In fact, we do not even need to broadcast new keys, they can be updated locally by each member, using a function which is difficult to invert, such as a cryptographically strong pseudo-random number generator (PRNG).

To handle that a member leaves, all the keys known to that member must be replaced. We also remove the internal node closest to the leaf corresponding to the leaving member. To distribute a new key for an internal node, we note that the leaving member knew the key corresponding to one of the two child nodes. Thus, we can encrypt the new key with the key of the child node the leaving member does not have. For the child which the leaving member is below, we will have distributed a new key accessible to everyone but the leaving member, so we use the new key of the child to encrypt the new key of the parent.

Building upon a similar structure as LKH, other schemes have been proposed. In LKH, we update keys with new keys which are unrelated to any other keys (except for our remark about joining members where we can apply a PRNG to update keys). To make schemes more efficient, we can use cryptographic functions to not only update a single key from an old one, but also to relate keys in different nodes of the tree. Works along these lines are One-way Function Trees by Sherman and McGrew [100] and ELK by Perrig, Song and Tygar [89]. Using keys which are related to each other via a cryptographic function is an idea we come back to in the context of the stateless subset cover schemes in Section 5.2.

Stateless

The largest class of stateless schemes are the schemes in the subset cover family, which we briefly discussed in Section 1.2. We discuss an alternate stateless approach in this section, and return to the subset cover paradigm in more detail in Section 5.2.

Boneh, Gentry and Waters [20] presented a stateless scheme based on so-called bilinear maps. We omit any details on what a bilinear map is, but we note that (cryptographically strong) bilinear maps have provided efficient solutions to many problems in cryptography. The scheme in [20] result in constant bandwidth to update a media key, and with constant amount of secret storage at each receiver. The drawback is that it requires each user to know $\mathcal{O}(n)$ public values. As the name implies, the public values need not be kept secret, but must still be stored and used in computations. As the memory requirements are a bit high, a modification to run $\mathcal{O}(\sqrt{n})$ instances of the scheme in parallel was also proposed in [20], resulting in $\mathcal{O}(\sqrt{n})$ bandwidth and $\mathcal{O}(\sqrt{n})$ (public) memory.

There are also a number of proposals that combine ideas from stateless and stateful schemes. As our Paper VII concerns making a stateless scheme stateful, we describe the related approaches in some detail.

Combining Stateless and Stateful Schemes

The stateless schemes in the subset cover paradigm are typically highly efficient when a small number of users are revoked, but inefficient when only a small fraction of users are members. The stateful schemes are more efficient when the number of members is small. Thus, one natural idea is to attempt to combine the two approaches. Such modifications typically yield a scheme that is stateful.

One approach presented by Chen *et al.* [29] is called Dynamic Subset Difference[4]. The core idea of [29] is to run a number of parallel, instances of a subset cover schemes (in this case, subset difference in particular, but the idea generalizes), with a relatively small n in each instance. When a new media key is distributed, the same key is distributed simultaneously in all instances. New users joining are assigned to the newest instance, if there are still unassigned positions in it. Otherwise, a new instance is created, and the user is assigned to it. Each individual member is a member (and user) in precisely one instance. When a member leaves, they are revoked in that instance, and thus cannot recover future media keys. If they become a member again, their status in that instance is switched back to being a member. If a too large fraction of users in an instance are revoked, the remaining members are reassigned to new instances, and the old instance is removed entirely. Reassignment is expensive and the authors evaluate several different policies for reassignment. The reassignment operation is the only stateful operation in [29].

Another approach by Jiang and Gong [57] is called hybrid broadcast encryption. Their proposed scheme is a more direct combination of a stateless and stateful scheme. The stateless scheme is used until some fixed number of users has been revoked, at which point a stateful round is run to update the key structure. A drawback with their scheme is that keys in the original scheme are replaced individually. They present specialized and more efficient key update methods for two particular subset cover schemes.

Finally, we get to our contribution! In Paper VII we propose a cheap mechanism to convert any subset cover scheme into a stateful scheme. This is done by adding a *state key* in addition to the media key, that is updated whenever the media key is updated. By encrypting the broadcast of a new media key (and state key) using a combination of the key for a selected subset and the current state key, the subset cover process need only avoid subsets containing the revoked users who left since the last update of the state key. This results in significantly better bandwidth performance than in subset cover schemes, but two drawbacks is that the resulting scheme is stateful and that it is no longer secure against a collusion of users.

With this we conclude our overview of approaches to constructing broadcast encryption schemes. We return to a more detailed discussion on subset cover schemes in Section 5.2, but first we discuss how to identify dishonest members.

Traitor Tracing

A topic related to broadcast encryption is that of traitor tracing. In many of the media distribution scenarios where broadcast encryption can be used, a real-world attacker would want to build unlicensed decoding equipment. For instance in the

[4]Subset Difference is the most influential scheme in the subset cover family.

context of cable TV, so-called pirate decoders are a problem for the cable networks. If broadcast encryption is used to protect the content, then such a pirate decoder would have keys corresponding to one or more users in the system. By revoking those keys, the decoder would cease to function. But how do we know what keys were used to build the decoder?

Traitor tracing is the problem of efficiently discovering what keys were used to build an unlicensed decoder. The assumption is that the sender can get hold of an unlicensed device. The sender can then create broadcasts targeted at specific sets of users and test if the decoder can successfully decrypt those. A naive solution would be to create n test broadcasts, one for each user in system, and test them in turn. This may take too much time to be practically usable, however, imagine sending a million test signals to a box and waiting to get to the single one that works. By looking at the properties of how keys are distributed, it is often possible to find more efficient testing methods which can find the user with far fewer tests, typically $\mathcal{O}(\log n)$.

The traditional traitor tracing notions relate primarily to unlicensed hardware or software decoders. There has been work to extend this model to also cover other types of illegitimate use. These include techniques which cause members to decrypt content slightly differently, slight differences that can then be used to identify the member from content uploaded to a pirate site [58].

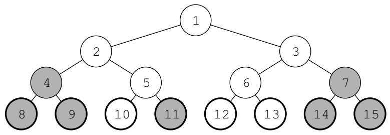
Having finished our introduction of broadcast encryption and discussion on traitor tracing, we return to the subject of subset cover schemes. These are the most common and influential stateless broadcast encryption schemes.

5.2 Subset Cover

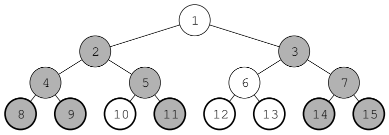
The subset cover framework is a very general framework for constructing stateless broadcast encryption schemes, and was introduced by Naor, Naor, and Lotspiech [83]. We call a scheme constructed following the framework a *subset cover scheme*. The core idea is that there is a fixed set of keys used to distribute the media keys. As the subsets are fixed, subset cover keys require the number of users n to be constant, fixed when the system is initialized[5]. Each key is known to the sender and to some subset of the users. We can associate each key with the subset of the users who know the key. We denote the keys by $k_1, k_2, \ldots, k_m$, and the associated sets by $S_1, S_2, \ldots, S_m$.

To distribute a media key K_M, the sender selects a number of subsets such that each member is in at least one selected subset and no revoked user is in any selected subset. Such a selection is known as a *subset cover*. The sender's message contains the media key encrypted under the key of each subset in the cover. The message also contains information on what subsets were used in the cover, to inform members which part of the message to decrypt. We write encryption of m under

[5]There are a few exceptions without this requirement, e.g. [56].



(a) Complete Subtree



(b) Subset Difference

Figure 5.2: Two subset cover schemes, both covering the same member set (sets used marked in grey).

key k as $\mathsf{E}(k, m)$. If the sender uses subsets S_1, S_4, S_7 in the cover, her message would be $1, 4, 7, \mathsf{E}(k_1, K_M), \mathsf{E}(k_4, K_M), \mathsf{E}(k_7, K_M)$.

From this description, we see that the size of the broadcast is proportional to the number of subsets used in the cover. To minimize the bandwidth a sender wants to select as few subsets as possible while still covering all members and not covering any revoked users. As it turns out, finding the smallest possible cover is a problem we cannot solve efficiently in general (it is NP-hard [62]). This difficulty applies to a scheme with the subsets S_i used in the scheme selected in an arbitrary fashion. Most schemes have subsets S_i such that it is easy to compute the smallest possible cover for any choice of revoked users.

A number of subset cover schemes have been proposed in the literature. Along with the definition of the subset cover framework, two schemes were also presented in [83]: Complete Subtree (CS) and Subset Difference (SD). In both schemes, as with many other schemes, the structure of the subsets used can be described by thinking of the users as positioned in the leafs of a balanced binary tree. A key difference here compared to the stateful LKH scheme we discussed in Section 5.1 is that all *users* are part of the tree and know some of the keys, not only the current *members*.

In the complete subtree scheme, there is a key associated with each node in the binary tree. This construction is similar to that of LKH, and each user knows the keys of all the nodes on the path from their leaf to the root. We illustrate in Figure 5.2a a complete subtree scheme where the current members are the users at nodes 8, 9, 11, 14, and 15. To cover this set, three subsets are used, corresponding to the subtrees rooted in nodes 4, 11, and 7. In the complete subtree scheme, each user u_i has to store $\log n$ keys as her set of secrets s_i, and the size of a cover (and thus bandwidth) can be shown to be $\min(r \log n/r, m)$.

The subset difference (SD) scheme is the most influential broadcast encryption scheme. In this scheme, the subsets in the scheme are the subsets on the form "all users below node x, except for those who are also below node y", for all x and y. This family of subsets is significantly larger than the one in the complete subtree scheme, and results in much lower bandwidth. In Figure 5.2b we illustrate an SD example where the current members are the same as in our complete subtree example. This time, only two subsets are needed to cover the members: all below

2 but not 10, and all below 3 but not 6. In general, the sender needs only to select at most $\min(2r-1, m)$ subsets in the cover.

In Section 5.1 we said that a user u_i stores a set of secrets s_i, that can be used to derive her keys. In the SD scheme, each user belongs to $\mathcal{O}(n)$ subsets. Storing that many keys is infeasible in many settings, e.g. cable TV and BluRay players, where the set-top box or player must be as cheap as possible. To solve this, the keys are related to each other by a cryptographic function, a pseudo-random sequence generator (PRSG). This allows each user to only store $\mathcal{O}(\log^2 n)$ secrets, and still be able to generate all the keys for the subsets to which she belongs. The secrets here are so-called seeds of the PRSG, a construction that the SD scheme has in common with most other subset cover schemes. We omit details on how the derivation of keys works, for details see [83, 5].

Since the publication of [83], a large number of subset cover schemes have been proposed. We do not go into detail on such constructions. Many of them use a similar key structure as the SD schemes, examples include Layered Subset Difference [49], and Stratified Subset Difference [47] which yield constructions where a user need only store $\mathcal{O}(\log n)$ keys with only slightly higher bandwidth requirements than in the SD scheme. Another scheme is the Punctured Interval scheme [56], which also has the property that new users can be added to the scheme after initialization has been performed.

Having discussed some subset cover schemes, we now turn to the question of whether it is possible to construct more efficient schemes within the subset cover framework.

Impossibility Results

All of the subset cover schemes we have presented so far have almost the same worst-case performance in terms of the numbers of sets needed in a cover. In particular, they all need to select $\min(cr, m)$ sets in the worst case, for a small constant c depending on the scheme and its parameters. Thus, one may naturally ask: could we do better?

In [84], the full version of [83], a lower bound on the bandwidth required in a broadcast encryption scheme was proven by a simple information-theoretic argument. It was stated that the message sent from the sender, implicitly or explicitly, contains information on who is a member. This means that the message, at a minimum, must contain sufficient information to encode the membership information. There are $\binom{n}{r}$ ways to revoke r users out of n, so the message must be of a size at least $\log\binom{n}{r} \approx r\log n$. As the scheme selects $\mathcal{O}(r)$ subsets for small r, the overhead is $\mathcal{O}(r \cdot \text{keysize})$, where keysize is the size of the media key.

In Paper VI, we prove that schemes which use reasonably sized[6] sets of secrets s_i and which use a PRSG for key derivation, must use $\Omega(r)$ subsets in the cover,

[6]Here, "reasonably sized" means polynomial, which is much higher than what is normally considered reasonable.

and thus have a bandwidth of $\Omega(r \cdot \text{keysize})$. This shows that the bandwidth of the original SD scheme is in fact optimal, up to a small constant. In the same paper, we also prove that for some number of revoked users, the scheme must select $\Omega(n/\log s)$ sets, where s is the size of the set of secrets s_i stored at each user. Finally, we also prove that for sufficiently large r, the scheme must select $\Omega(m)$ sets, meaning that the performance degrades to that of the naive scheme.

Our impossibility results show that to significantly outperform the SD scheme, one must change one of the conditions used in the impossibility result. This may entail using a more advanced mechanism than a PRSG to compute keys, or working outside of the subset cover framework. Another option is to slightly modify the problem we solve, an approach we discuss next.

5.3 Related Problems

Apart from the normal broadcast encryption model, related problems and models have been proposed. One of these is the case of attribute-based encryption and its application to broadcast. In this setting, each user is associated with some set of attributes. A broadcast can then be targeted to specific attributes, or combinations of attributes. This can allow for efficient revocations of users based on e.g. properties of their cable-TV subscriptions which can be encoded as attributes. Using attribute-encryption in broadcast encryption applications was discussed by Goyal *et al.* [48].

Free Riders and Weakened Security Models

For subset cover based schemes, it may be the case that allowing a few revoked users to access the media key can reduce the bandwidth requirements. To see this, consider the case where a very large subset is unusable due to a small number of revoked users being in the subset, forcing the scheme to select many small sets to cover the members. An idea for an optimization is to allow the scheme to sometimes also cover revoked members, allowing them to access the content.

This can be acceptable in commercial settings for media streaming, a pirate decoder which can access content at unpredictable times is not something that most users would buy. The relaxed problem where revoked users can access content sometimes is known as broadcast encryption with free riders, a setting first discussed by Abdalla, Shavitt, and Wool [1].

One scheme in the setting with free riders is due to Adelsbach and Greveler [2]. They proposed two schemes, which essentially distribute the key piece-wise[7] to multiple sets. Each member is guaranteed to get enough pieces that they can recover the key, while most revoked users receive too few pieces to do the same. A drawback of their scheme is that it is not secure against collusions. Another result is by Ramzan and Woodruff [92] who gave algorithms to optimally select the

[7]Either bit by bit or using secret sharing.

revoked users who are allowed to receive the content to save as much bandwidth as possible for a given number of free riders.

Another weakening of properties to build a more efficient system is presented in our Paper VII. There, the system prevents all revoked users from accessing content, so there are no free riders. The property we weaken is that we lose security against collusions. The transformation presented there is vulnerable to a leaving member colluding with a previously revoked user. With such a collusion, it may be that the pair would be able to continue decrypting the broadcast.

With this, we are now ready to conclude this section by summarizing our results and discussing the main real-world application of broadcast encryption.

5.4 Contributions

In this thesis, we present two results on broadcast encryption, in particular on subset cover based broadcast encryption schemes. The work here is focused on the bandwidth requirements of such systems.

The first contribution, Paper VII, is the introduction of a state-key to transform a subset cover scheme into what we call a stateful subset cover scheme to save bandwidth. The state key is distributed along-side the media encryption key, so only the current members have the state key. To distribute a new media encryption key (and state key), the sender encrypts using a combination of the current state key and the key associated with the selected subsets. This allows us to use larger subsets which may cover some revoked user, as those revoked users do not know the current state key. This increased efficiency comes at a price as the security properties of a stateful subset cover scheme are worse than those of the original scheme. In commercial settings, this may still be acceptable, as e.g. a pirate decoder which works only sometimes is of relatively little value. The drawback can also be mitigated by periodically distributing a new media key and state key using the original subset cover scheme we build upon, doing so revokes colluding users until one of them becomes a member again.

Our second contribution, Paper VI, is more on the theoretical side. We prove three lower bounds on the bandwidth a subset cover scheme must use. Our bounds apply to schemes using a PRSG to compute keys, the most common design choice. We show that the original SD scheme was close to being optimal.

5.5 Real-world Applications

The largest deployment of broadcast encryption schemes is to protect the content of BluRay discs (and also the now-dead HD-DVD format). These use a copy-protection method called Advanced Access Content System (AACS). One of the core components of this method is a subset cover based broadcast encryption scheme, more specifically the subset difference (SD) scheme.

Each manufacturer of BluRay players must have a license from the AACS Licensing Administrator (AACS LA). When licensed, they receive sets of secrets corresponding to users in an SD scheme. The manufacturer may either let each user in the SD scheme correspond to a specific product (i.e., a BluRay player), or it may choose to let multiple products be the same user in the SD scheme. The set of secrets corresponding to the user is embedded in the hardware.

The AACS LA acts as a sender in this system. When a BluRay disc is produced, the AACS LA produces a message in the SD system with a new random media key unique to the disc. The message and the media key is given to the company creating the disc, which encrypts the content using the media key, and places the SD message at the beginning of the disc. Thus, the content on a BluRay disc is the same as the broadcast of encrypted media in a broadcast encryption scheme: first a message with a media key, and then the content encrypted with the key.

A licensed player can recover the media key from the message on the disc and decrypt the content. If it is discovered that the secrets corresponding to some BluRay player have leaked, that “user” can be revoked. This means that such players are unable to play any BluRay discs manufactured after when the revocation occurred (as the player will then be revoked in the SD scheme and cannot decrypt the media key).

The AACS standard also has support for traitor tracing. Firstly, if an unlicensed player appears on the market, normal traitor tracing methods can be applied to discover what user to revoke to disable the device from decrypting future discs. Secondly, the AACS standard also provides mechanisms to make players decrypt content slightly differently, such that content uploaded to pirate sites can be traced back to a player model.

Errata for Included Publications

At least one of the publications included in this thesis contained at least one error at the time of publication. We included the version as published in this thesis, and note the known error here:

- In Paper VII on page 7 (184 in the paper), $S_{3,14}$ should be $S_{3,15}$.

Bibliography

[1] Michel Abdalla, Yuval Shavitt, and Avishai Wool. Key management for restricted multicast using broadcast encryption. *IEEE/ACM Trans. Netw.*, 8 (4):443–454, 2000.

[2] André Adelsbach and Ulrich Greveler. A broadcast encryption scheme with free-riders but unconditional security. In Reihaneh Safavi-Naini and Moti Yung, editors, *DRMTICS*, volume 3919 of *Lecture Notes in Computer Science*, pages 246–257. Springer, 2005. ISBN 3-540-35998-2.

[3] Alexa. Top 500 global sites, 2011. URL `http://www.alexa.com/topsites`. Accessed 2011-03-18.

[4] Anonymous. Did you watch porn, 2010. URL `http://didyouwatchporn.com/`. Accessed 2010-11-19.

[5] Nuttapong Attrapadung and Hideki Imai. Graph-decomposition-based frameworks for subset-cover broadcast encryption and efficient instantiations. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 100–120. Springer, 2005. ISBN 3-540-30684-6.

[6] Per Austrin and Gunnar Kreitz. Lower bounds for subset cover based broadcast encryption. In Serge Vaudenay, editor, *AFRICACRYPT*, volume 5023 of *Lecture Notes in Computer Science*, pages 343–356. Springer, 2008. ISBN 978-3-540-68159-5.

[7] József Balogh, János A. Csirik, Yuval Ishai, and Eyal Kushilevitz. Private computation using a PEZ dispenser. *Theor. Comput. Sci.*, 306(1-3):69–84, 2003.

[8] L. David Baron. Preventing attacks on a user's history through CSS :visited selectors. URL `http://dbaron.org/mozilla/visited-privacy`. Accessed 2010-11-19.

[9] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 75–88. ACM, 2008. ISBN 978-1-59593-810-7.

[10] Mayank Bawa, Hrishikesh Deshpande, and Hector Garcia-Molina. Transience of peers & streaming media. *Computer Communication Review*, 33 (1):107–112, 2003.

[11] Donald Beaver. Perfect privacy for two-party protocols. In J. Feigenbaum and M. Merritt, editors, *Proceedings of DIMACS Workshop on Distributed Computing and Cryptology*, volume 2, pages 65–77. American Mathematical Society, 1989.

[12] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *STOC*, pages 503–513. ACM, 1990.

[13] Amos Beimel. On private computation in incomplete networks. *Distributed Computing*, 19(3):237–252, 2007.

[14] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 257–266. ACM, 2008. ISBN 978-1-59593-810-7.

[15] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10. ACM, 1988.

[16] Shimshon Berkovits. How to broadcast a secret. In Donald W. Davies, editor, *EUROCRYPT 1991*, volume 547 of *LNCS*, pages 535–541. Springer, 1991. ISBN 3-540-54620-0.

[17] Markus Bläser, Andreas Jakoby, Maciej Liskiewicz, and Bodo Manthey. Private computation: k-connected versus 1-connected networks. *J. Cryptology*, 19(3):341–357, 2006.

[18] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008. ISBN 978-3-540-88312-8.

[19] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

[20] Dan Boneh, Craig Gentry, and Brent Waters. Collusion resistant broadcast encryption with short ciphertexts and private keys. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 258–275. Springer, 2005.

[21] Jake Brutlag. Speed matters for Google web search, 2009. URL `http://code.google.com/speed/files/delayexp.pdf`. Accessed 2011-03-20.

[22] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *19th USENIX Security Symposium*, Washington, DC, USA, August 2010.

[23] Eric Butler. Firesheep. URL `http://codebutler.com/firesheep`. Accessed 2010-11-19.

[24] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. Internet Group Management Protocol, Version 3. RFC 3376 (Proposed Standard), October 2002. URL `http://www.ietf.org/rfc/rfc3376.txt`.

[25] CERT. Smurf IP denial-of-service attacks. Technical Report CA-1998-01, CERT, 1998. URL `http://www.cert.org/advisories/CA-1998-01.html`.

[26] CERT. Malicious HTML tags embedded in client web requests. Technical Report CA-2000-02, CERT, 2000. URL `http://www.cert.org/advisories/CA-2000-02.html`.

[27] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptology*, 1(1):65–75, 1988.

[28] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *STOC*, pages 11–19. ACM, 1988.

[29] Weifeng Chen, Zihui Ge, Chun Zhang, James F. Kurose, and Donald F. Towsley. On dynamic subset difference revocation scheme. In Nikolas Mitrou, Kimon P. Kontovasilis, George N. Rouskas, Ilias Iliadis, and Lazaros F. Merakos, editors, *NETWORKING*, volume 3042 of *Lecture Notes in Computer Science*, pages 743–758. Springer, 2004. ISBN 3-540-21959-5.

[30] Benny Chor, Mihály Geréb-Graus, and Eyal Kushilevitz. On the structure of the privacy hierarchy. *J. Cryptology*, 7(1):53–60, 1994.

[31] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults (extended abstract). In *FOCS*, pages 383–395. IEEE, 1985.

[32] Benny Chor and Yuval Ishai. On privacy and partition arguments. *Inf. Comput.*, 167(1):2–9, 2001.

[33] Benny Chor and Eyal Kushilevitz. A zero-one law for boolean privacy. *SIAM J. Discrete Math.*, 4(1):36–47, 1991.

[34] Steve Christey and Robert A. Martin. Vulnerability type distributions in CVE. Technical report, The MITRE Corporation, 2007. URL `http://cwe.mitre.org/documents/vuln-trends.html`.

[35] Nicolas Christin, Sally S. Yanagihara, and Keisuke Kamataki. Dissecting one click frauds. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 15–26. ACM, 2010. ISBN 978-1-4503-0245-6.

[36] Yang-Hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast. In *SIGMETRICS*, pages 1–12, 2000.

[37] Bram Cohen. The BitTorrent protocol specification. BEP 3 (Standard), 2008. URL `http://www.bittorrent.org/beps/bep_0003.html`.

[38] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009. ISBN 978-3-642-00467-4.

[39] DE-CIX. De-cix traffic statistics, 2011. URL `https://www.de-cix.net/content/network/statistics.html`. Accessed 2011-04-04.

[40] Bert den Boer. More efficient match-making and satisfiability: *the Five Card Trick*. In *EUROCRYPT*, pages 208–217, 1989.

[41] Hrishikesh Deshpande, Mayank Bawa, and Hector Garcia-Molina. Streaming live media over peers. Technical Report 2002-21, Stanford InfoLab, 2002. URL `http://ilpubs.stanford.edu:8090/863/`.

[42] Edward W. Felten and Michael A. Schneider. Timing attacks on web privacy. In *ACM Conference on Computer and Communications Security*, pages 25–32, 2000.

[43] Matthew K. Franklin and Moti Yung. Secure hypergraphs: Privacy from partial broadcast. *SIAM J. Discrete Math.*, 18(3):437–450, 2004.

[44] Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, and Maxime Monod. Boosting gossip for live streaming. In *Peer-to-Peer Computing*, pages 1–10. IEEE, 2010. ISBN 978-1-4244-7141-6.

[45] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4): 469–472, 1985.

[46] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229. ACM, 1987.

[47] Michael T. Goodrich, Jonathan Z. Sun, and Roberto Tamassia. Efficient tree-based revocation in groups of low-state devices. In Matthew K. Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 511–527. Springer, 2004. ISBN 3-540-22668-0.

[48] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 89–98. ACM, 2006.

[49] Dani Halevy and Adi Shamir. The LSD broadcast encryption scheme. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 47–60. Springer, 2002. ISBN 3-540-44050-X.

[50] Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *J. Cryptology*, 13(1):31–60, 2000.

[51] Yan Huang, Tom Z.J. Fu, Dah-Ming Chiu, John C.S. Lui, and Cheng Huang. Challenges, design and analysis of a large-scale P2P-VoD system. *SIGCOMM Comput. Commun. Rev.*, 38(4):375–388, 2008. ISSN 0146-4833.

[52] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting browsers from DNS rebinding attacks. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 421–431. ACM, 2007. ISBN 978-1-59593-703-2.

[53] Markus Jakobsson, Ari Juels, and Jacob Ratkiewicz. Remote harm-diagnostics. URL `http://www.ravenwhite.com/files/rhd.pdf`. Accessed 2010-11-19.

[54] Artur Janc and Lukasz Olejnik. Web browser history detection as a real-world privacy threat. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *ESORICS*, volume 6345 of *Lecture Notes in Computer Science*, pages 215–231. Springer, 2010. ISBN 978-3-642-15496-6.

[55] Artur Janc and Lukasz Olejnik. What the internet knows about you, 2010. URL `http://www.wtikay.com/`. Accessed 2010-11-19.

[56] Nam-Su Jho, Jung Yeon Hwang, Jung Hee Cheon, Myung-Hwan Kim, Dong Hoon Lee, and Eun Sun Yoo. One-way chain based broadcast encryption schemes. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 559–574. Springer, 2005. ISBN 3-540-25910-4.

[57] Shaoquan Jiang and Guang Gong. Hybrid broadcast encryption and security analysis. Cryptology ePrint Archive, Report 2003/241, 2003. `http://eprint.iacr.org/`.

[58] Hongxia Jin and Jeffery Lotspiech. Renewable traitor tracing: A trace-revoke-trace system for anonymous attack. In Joachim Biskup and Javier Lopez, editors, *ESORICS*, volume 4734 of *Lecture Notes in Computer Science*, pages 563–577. Springer, 2007. ISBN 978-3-540-74834-2.

[59] Mattias Johansson, Gunnar Kreitz, and Fredrik Lindholm. Stateful subset cover. In Jianying Zhou, Moti Yung, and Feng Bao, editors, *ACNS*, volume 3989 of *Lecture Notes in Computer Science*, pages 178–193, 2006. ISBN 3-540-34703-8.

[60] Samy Kamkar. How I met your girlfriend, 2010. URL `http://samy.pl/bh10/`. Accessed 2011-03-11.

[61] Samy Kamkar. phpwn: Attacking sessions and pseudo-random numbers in PHP. In *Proceedings of Blackhat USA*, 2010.

[62] R. M. Karp. Reducibility Among Combinatorial Problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[63] Joe Kilian, Eyal Kushilevitz, Silvio Micali, and Rafail Ostrovsky. Reducibility and completeness in private computations. *SIAM J. Comput.*, 29(4): 1189–1208, 2000.

[64] Amit Klein. DOM based cross site scripting or XSS of the third kind. Technical report, Web Application Security Consortium, 2005. URL `http://www.webappsec.org/projects/articles/071105.shtml`.

[65] Dejan Kostic, Adolfo Rodriguez, Jeannie R. Albrecht, and Amin Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In Michael L. Scott and Larry L. Peterson, editors, *SOSP*, pages 282–297. ACM, 2003. ISBN 1-58113-757-5.

[66] Gunnar Kreitz. A zero-one law for secure multi-party computation with ternary outputs. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *Lecture Notes in Computer Science*, pages 382–399. Springer, 2011. ISBN 978-3-642-19570-9.

[67] Gunnar Kreitz. A zero-one law for secure multi-party computation with ternary outputs (full version). Cryptology ePrint Archive, Report 2011/002, 2011. `http://eprint.iacr.org/`.

[68] Gunnar Kreitz, Mads Dam, and Douglas Wikström. Practical private information aggregation in large networks. In *NordSec*, 2010.

[69] Gunnar Kreitz and Fredrik Niemelä. Spotify – large scale, low latency, P2P music-on-demand streaming. In *Peer-to-Peer Computing*, pages 1–10. IEEE, 2010. ISBN 978-1-4244-7141-6.

[70] M. V. N. Ashwin Kumar, Pranava R. Goundan, K. Srinathan, and C. Pandu Rangan. On perfectly secure cmmunication over arbitrary networks. In *PODC*, pages 193–202, 2002.

[71] Eyal Kushilevitz. Privacy and communication complexity. *SIAM J. Discrete Math.*, 5(2):273–284, 1992.

[72] Guanfeng Liang and Ben Liang. Jitter-free probability bounds for video streaming over random VBR channel. In *Proc. of QShine'06*, page 6, New York, NY, USA, 2006. ACM. ISBN 1-59593-537-1.

[73] X. Liao, H. Jin, Y. Liu, L. M. Ni, and D. Deng. AnySee: Peer-to-peer live streaming. In *Proc. of IEEE INFOCOM'06*, pages 1–10, 2006.

[74] Fangming Liu, Shijun Shen, Bo Li, Baochun Li, Hao Yin, and Sanli Li. Novasky: Cinematic-quality VoD in a P2P storage cloud. In *Proc. of IEEE INFOCOM'11*, 2011. To appear.

[75] F. Ljunggren, T. Okubo, R. Lamb, and J. Schlyter. DNSSEC Practice Statement for the Root Zone KSK Operator, May 2010. URL `https://www.iana.org/dnssec/icann-dps.txt`.

[76] Christoph Lucas, Dominik Raub, and Ueli M. Maurer. Hybrid-secure MPC: trading information-theoretic robustness for computational privacy. In Andréa W. Richa and Rachid Guerraoui, editors, *PODC*, pages 219–228. ACM, 2010. ISBN 978-1-60558-888-9.

[77] Nazanin Magharei, Rejaie Rejaie, and Yang Guo. Mesh or multiple-tree: A comparative study of live P2P streaming approaches. In *Proc. of IEEE INFOCOM'07*, pages 1424–1432, 2007.

[78] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay – secure two-party computation system. In *USENIX Security Symposium*, pages 287–302. USENIX, 2004.

[79] Mog. About Mog inc, 2011. URL `http://mog.com/about`. Accessed 2011-03-05.

[80] Jacob Jan-David Mol, Arno Bakker, Johan A. Pouwelse, Dick H. J. Epema, and Henk J. Sips. The design and deployment of a bittorrent live video streaming solution. In *ISM*, pages 342–349. IEEE Computer Society, 2009. ISBN 978-0-7695-3890-7.

[81] Mozilla. Geolocation in firefox, 2010. URL `http://www.mozilla.com/en-US/firefox/geolocation/`. Accessed 2011-03-12.

[82] Jörn Müller-Quade and Dominique Unruh. Long-term security and universal composability. *J. Cryptology*, 23(4):594–671, 2010.

[83] Dalit Naor, Moni Naor, and Jeffery Lotspiech. Revocation and tracing schemes for stateless receivers. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 41–62. Springer, 2001. ISBN 3-540-42456-3.

[84] Dalit Naor, Moni Naor, and Jeffery Lotspiech. Revocation and tracing schemes for stateless receivers. *Electronic Colloquium on Computational Complexity (ECCC)*, 2002(043), 2002.

[85] Noam Nisan and Amir Ronen. Algorithmic mechanism design (extended abstract). In *STOC*, pages 129–140, 1999.

[86] NIST. *Advanced Encryption Standard (AES) (FIPS PUB 197)*. National Institute of Standards and Technology, November 2001.

[87] OWASP. OWASP top 10, 2010. URL `http://owasptop10.googlecode.com/files/OWASP\%20Top\%2010\%20-\%202010.pdf`. Accessed 2010-11-19.

[88] Venkata N. Padmanabhan, Helen J. Wang, Philip A. Chou, and Kunwadee Sripanidkulchai. Distributing streaming media content using cooperative networking. In *NOSSDAV*, pages 177–186. ACM, 2002.

[89] Adrian Perrig, Dawn Xiaodong Song, and J. D. Tygar. ELK, a new protocol for efficient large-group key distribution. In *IEEE Symposium on Security and Privacy*, pages 247–, 2001.

[90] Fabio Picconi and Laurent Massoulié. Is there a future for mesh-based live video streaming? In Klaus Wehrle, Wolfgang Kellerer, Sandeep K. Singhal, and Ralf Steinmetz, editors, *Peer-to-Peer Computing*, pages 289–298. IEEE Computer Society, 2008. ISBN 978-0-7695-3318-6.

[91] Johan A. Pouwelse, Pawel Garbacki, Jun Wang, Arno Bakker, Jie Yang, Alexandru Iosup, Dick H. J. Epema, Marcel J. T. Reinders, Maarten van Steen, and Henk J. Sips. Tribler: a social-based peer-to-peer system. *Concurrency and Computation: Practice and Experience*, 20(2):127–138, 2008.

[92] Zulfikar Ramzan and David P. Woodruff. Fast algorithms for the free riders problem in broadcast encryption. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 308–325. Springer, 2006. ISBN 3-540-37432-9.

[93] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport Layer Security (TLS) Renegotiation Indication Extension. RFC 5746 (Proposed Standard), February 2010. URL `http://www.ietf.org/rfc/rfc5746.txt`.

[94] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[95] Roberto Roverso, Amgad Naiem, Mohammed Reda, Mohammed El-Beltagy, Sameh El-Ansary, Nils Franzen, and Seif Haridi. On the feasibility of centrally-coordinated peer-to-peer live streaming. Technical report, Peerialism, 2010. URL `http://www.peerialism.com/#whitepapers`.

[96] Jesse Ruderman. Bug 57351 - css on a:visited can load an image and/or reveal if visitor been to a site. URL `https://bugzilla.mozilla.org/show_bug.cgi?id=57351`. Accessed 2010-11-19.

[97] Sandvine. Fall 2010 global internet phenomena report, 2010. URL `http://www.sandvine.com/downloads/documents/2010%20Global%20Internet%20Phenomena%20Report.pdf`. Accessed 2011-03-18.

[98] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[99] Eric Sheridan. OWASP CSRFGuard project, 2008. URL `http://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project`. Accessed 2011-03-24.

[100] Alan T. Sherman and David A. McGrew. Key establishment in large dynamic groups using one-way function trees. *IEEE Trans. Software Eng.*, 29(5):444–458, 2003.

[101] Spotify. Music catalogue info, 2011. URL `https://www.spotify.com/se/about/music-catalogue-info/`. Accessed 2011-03-05.

[102] Frank Stajano and Ross J. Anderson. The cocaine auction protocol: On the power of anonymous broadcast. In Andreas Pfitzmann, editor, *Information Hiding*, volume 1768 of *Lecture Notes in Computer Science*, pages 434–447. Springer, 1999. ISBN 3-540-67182-X.

[103] D.A. Tran, K.A. Hua, and T. Do. ZIGZAG: an efficient peer-to-peer scheme for media streaming. In *Proc. of IEEE INFOCOM'03*, pages 1283–1292, 2003.

[104] Voddler. About voddler, 2011. URL `http://www.voddler.com/about/`. Accessed 2011-03-05.

[105] Debby M. Wallner, Eric J. Harder, and Ryan C. Agee. Key management for multicast: Issues and architectures. Internet Request for Comment RFC 2627, Internet Engineering Task Force, 1999. URL `http://www.ietf.org/rfc/rfc2627.txt`.

[106] Chung Kei Wong, Mohamed G. Gouda, and Simon S. Lam. Secure group communications using key graphs. In *SIGCOMM*, pages 68–79, 1998.

[107] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE, 1982.

[108] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167. IEEE, 1986.

[109] Michal Zalewski. Browser security handbook, 2008. URL `http://code.google.com/p/browsersec/wiki/Main`. Accessed 2011-03-24.

[110] Xinyan Zhang, Jiangchuan Liu, Bo Li, and Tak-Shing Peter Yum. CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming. In *Proc. of IEEE INFOCOM'05*, pages 2102 – 2111, 2005.

I

Spotify – Large Scale, Low Latency, P2P Music-on-Demand Streaming

Gunnar Kreitz
KTH – Royal Institute of Technology, and Spotify
Stockholm, Sweden
Email: gkreitz@kth.se

Fredrik Niemelä
KTH – Royal Institute of Technology, and Spotify
Stockholm, Sweden
Email: niemela@kth.se

***Abstract*—Spotify is a music streaming service offering low-latency access to a library of over 8 million music tracks. Streaming is performed by a combination of client-server access and a peer-to-peer protocol. In this paper, we give an overview of the protocol and peer-to-peer architecture used and provide measurements of service performance and user behavior.**

The service currently has a user base of over 7 million and has been available in six European countries since October 2008. Data collected indicates that the combination of the client-server and peer-to-peer paradigms can be applied to music streaming with good results. In particular, 8.8% of music data played comes from Spotify's servers while the median playback latency is only 265 ms (including cached tracks). We also discuss the user access patterns observed and how the peer-to-peer network affects the access patterns as they reach the server.

I. Introduction

Spotify is a streaming music service using peer-to-peer techniques. The service has a library of over 8 million tracks, allowing users to freely choose tracks they wish to listen to and to seek within tracks. Data is streamed from both servers and a peer-to-peer network. The service launched in October 2008 and now has over 7 million users in six European countries.

The service is offered in two versions: a free version with advertisement, and a premium, pay-per-month, version. The premium version includes some extra features such as the option to stream music at a higher bitrate, and to synchronize playlists for offline usage. Both versions of the service allow unlimited streaming, and a large majority of users are on the free version. The music catalog is the same for both free and premium users with the exception of some pre-releases exclusive to premium users. However, due to licensing restrictions, the tracks accessible to a user depends on the user's home country.

One of the distinguishing features of the Spotify client is its low playback latency. The median latency to begin playback of a track is 265 ms. The service is not web-based, but instead uses a proprietary client and protocol.

A. Related Services

There are many different on-demand music streaming services offered today. To our knowledge, all such services but Spotify are web-based, using either Adobe Flash or a web browser plug-in for streaming. Furthermore, they are pure client-server applications without a peer-to-peer component. Among the more well-known such services are Napster, Rhapsody, and We7.

The application of peer-to-peer techniques to on-demand streaming is more prevalent when it comes to video-on-demand services. Such services include Joost, PPLive, and PPStream. These vary between supporting live streaming (usually of a large number of channels), video-on-demand access, or both. While there are many similarities between video-on-demand and music-on-demand streaming, there are also many differences; including user behavior, the size of streaming objects, and the number of objects offered for streaming.

A service offering on-demand streaming has many things in common with file-sharing applications. For instance, the mechanisms for locating peers in Spotify are similar to techniques from BitTorrent and Gnutella.

B. Related Work

Leveraging the scalability of peer-to-peer networks to perform media streaming is a well-studied area in the academic literature. Most such systems are concerned with live streaming, where viewers watch the same stream simultaneously. This setting is different in nature from the Spotify application, where a user has on-demand access to a large library of tracks.

The peer-to-peer live streaming literature can be roughly divided into two general approaches [1]: tree-based (e.g. [2]), and mesh-based (e.g. [3], [4]), depending on whether they maintain a tree structure in the overlay. While both techniques have their advantages, intuitively it seems that a mesh structure is a better fit for on-demand streaming applications.

There have been several studies measuring the performance and behavior of large peer-to-peer systems, describing and measuring both on-demand streaming [5], [6], and file-sharing [7], [8] protocols. We believe that there is high value in understanding how peer-to-peer techniques perform in today's networks.

Huang *et al.* [5] describe the PPLive video-on-demand streaming system, and also present measurements on its performance. To the best of our knowledge, their work is the only other detailed study of an on-demand streaming system of this size, with a peer-to-peer component. While there are naturally many similarities between PPLive and Spotify, there are also many differences, including the overlay structure and Spotify's focus on low latency techniques.

C. Our Contribution

In this paper we give an in-depth description and evaluation of Spotify. We discuss the general streaming protocol in Section II, and go into more details on the peer-to-peer parts in Section III.

Furthermore, in Section IV, we present and comment on data gathered by Spotify while operating its service. We give detailed measurements on many aspects of the service such as latency, stutter, and how much data is offloaded from the server by the peer-to-peer protocol. Some measurement data is also presented in Sections II and III; that data was collected as described in Section IV-A.

II. Spotify Overview

The Spotify protocol is a proprietary network protocol designed for streaming music. There are clients for OS X and Windows as well as for several smartphone platforms. The Windows version can also be run using Wine. The smartphone clients do not participate at all in the peer-to-peer protocol, but only stream from servers. Since the focus of this paper is the evaluation of peer-to-peer techniques we will ignore the smartphone clients in the remainder of this paper.

The clients are closed-source software available for free download, but to use a client, a Spotify user account is needed. Clients automatically update themselves, and only the most recent version is allowed to access the service.

The user interface is similar to those found in desktop mp3 players. Users can organize tracks into playlists which can be shared with others as links. Finding music is organized around two concepts: searching and browsing. A user can search for tracks, albums, or artists, and she can also browse—for instance, when clicking on an artist name, the user is presented with a page displaying all albums featuring that artist.

Audio streams are encoded using Ogg Vorbis with a default quality of q5, which has variable bitrate averaging roughly 160 kbps. Users with a premium subscription can choose (through a client setting) to instead receive Ogg Vorbis in q9 quality, averaging roughly 320 kbps. Both types of files are served from both servers and the peer-to-peer network. No re-encoding is done by peers, so a peer with the q9 version of a track cannot serve it to one wanting the q5 version.

When playing music, the Spotify client monitors the sound card buffers. If the buffers underrun, the client considers a *stutter* to have occurred. Stutters can be either due to network effects, or due to the client not receiving sufficient local resources to decode and decrypt data in a timely manner. Such local starvation is generally due to the client host doing other (predominantly I/O-intensive) tasks.

The protocol is designed to provide on-demand access to a large library of tracks and would be unsuitable for live broadcasts. For instance, a client cannot upload a track unless it has the whole track. The reason for this is that it simplifies the protocol, and removes the overhead involved with communicating what parts of a track a client has. The drawbacks are limited, as tracks are small.

While UDP is the most common transport protocol in streaming applications, Spotify instead uses TCP. Firstly, having a reliable transport protocol simplifies protocol design and implementation. Secondly, TCP is nice to the network in that TCP's congestion control is friendly to itself (and thus other applications using TCP), and the explicit connection signaling helps stateful firewalls. Thirdly, as streamed material is shared in the peer-to-peer network, the re-sending of lost packets is useful to the application.

Between a pair of hosts a single TCP connection is used, and the application protocol multiplexes messages over the connection. While a client is running, it keeps a TCP connection to a Spotify server. Application layer messages are buffered, and sorted by priority before being sent to the operating system's TCP buffers. For instance, messages needed to support interactive browsing are prioritized over bulk traffic.

A. Caching

Caching is important for two reasons. Firstly, it is common that users listen to the same track several times, and caching the track obviates the need for it to be re-downloaded. Secondly, cached music data can be served by the client in the peer-to-peer overlay. The cache can store partial tracks, so if a client only downloads a part of a track, that part will generally be cached. Cached content is encrypted and cannot be used by other players.

The default setting in the clients is for the maximum cache size to be at most 10% of free disk space (excluding the size of the cache itself), but at least 50 MB and at most 10 GB. The size can also be configured by the user to be between 1 and 100 GB, in 1 GB increments. This policy leads to most client installations having large caches (56% have a maximum size of 5 GB or more, and thus fit approximately 1000 tracks).

Cache eviction is done with a Least Recently Used (LRU) policy. Simulations using data on cache sizes and playback logs indicate that, as caches are large, the choice of cache eviction policy does not have a large effect on cache efficiency. This can be compared to the finding of Huang *et al.*[5] that the PPLive system gained much efficiency by changing from LRU to a more complex, weight-based evaluation process. However, in their setting, the objects in the cache are movies, and their caches are of a size such that a client can only cache one, or a few movies.

B. Random Access to a Track

The easiest case for a streaming music player is when tracks are played in a predictable order. Assuming sufficient bandwidth available, this allows the player to begin fetching data needed to play upcoming tracks ahead of time (*prefetching*). A more difficult, and perhaps interesting, case is when the user chooses a new track to be played, which we refer to as a *random access*. We begin by describing the random access case and then proceed to discuss prefetching in Section II-C.

Approximately 39% of playbacks in Spotify are by random access (the rest start because the current track finished, or because the user clicked the forward button to skip to the next track). Unless the client had the data cached, it makes an initial request to the server asking for approximately 15 seconds of music, using the already open TCP connection. Simultaneously, it searches the peer-to-peer network for peers who can serve the track, as described in Section III-C.

In most cases the initial request can be quickly satisfied. As the client already has a TCP connection to the server no 3-way handshake is needed. Common TCP congestion avoidance algorithms, such as TCP New Reno [9] and CUBIC [10], maintain a congestion window limiting how large bursts of data can be sent. The congestion window starts out small for a new connection and then grows as data is successfully sent. Normally, the congestion window (on the server) for the long-lived connection between client and server will be large as data will have been sent over it. This allows the server to quickly send much, or all, of the response to the latency-critical initial request without waiting for ACKs from the client.

The connection to the server is long-lived, but it is also bursty in nature. For instance, if a user is streaming from a popular album, a very large fraction of the traffic will be peer-to-peer traffic, and the connection to the server will then be almost unused. If the user then makes a random access playback there is a sudden burst of traffic. RFC 5681 [11] states that an implementation should reduce its congestion window if it has not sent data in an interval exceeding the retransmission timeout. Linux kernels can be configured to disable that reduction, and when Spotify did so the average playback latency decreased by approximately 50 ms. We remark that this was purely a server-side configuration change.

If a user jumps into the middle of a track (*seeks*), the client treats the request similarly to a random access, immediately requesting data from the server as described above. The Ogg Vorbis format offers limited support for seeking in a streaming environment, so Spotify adds a custom header to all their files to better support seeking.

C. Predictable Track Selection

Most playbacks (61%) occur in a predictable sequence, i.e. because the previous track played to its end, or because the user pressed the forward button. Clients begin prefetching the next track before the currently playing track has been played to completion. With prefetching there is a trade-off between cost and benefit. If clients begin prefetching too late the prefetching may not be sufficient to allow the next track to immediately start playing. If they prefetch too early the bandwidth may be wasted if the user then makes a random request.

The clients begin searching the peer-to-peer network and start downloading the next track when 30 seconds or less remain of the current track. When 10 seconds or less remain of the current track, the client prefetches the beginning of the next track from the server if needed.

We did not have data to directly measure how good the choice of these parameters are. But, we can measure how often a user who listens to a track for the duration of the track minus t seconds continues to the next track. If a user seeks within a track, the measured event does not correspond to playback coming within t seconds of the end of the track. For $t = 10, 30$, the next scheduled track was played in 94%, and 92% of cases, respectively. This indicates that the choice of parameters is reasonable, possibly a bit conservative.

During a period of a few weeks, all clients had a bug where prefetching of the next track was accidentally disabled. This allows us to directly measure the effects of prefetching on the performance of the system. During a week when prefetching was disabled the median playback latency was 390 ms, compared with the median latency over the current measurement period of 265 ms. Furthermore, the fraction of track playbacks in which stutter occurred was 1.8%, compared to the normal rate of 1.0%.

D. Regular Streaming

While streaming, clients avoid downloading data from the server unless it is necessary to maintain playback quality or keep down latency. As discussed in Section II-B, when the user makes a random access, an initial request for data is sent to the server. Clients make local decisions about where to stream from depending on the amount of data in their play-out buffers. The connection to the server is assumed to be more reliable than peer-connections, so if a client's buffer levels are low, it requests data from the server. As long as the client's buffers are sufficiently full and there are peers to stream from, the client only streams from the peer-to-peer network.

There is also an "emergency mode" where, if buffers become critically low (less than 3 seconds of audio buffered during playback), the client pauses uploading data to its peers. The reason for this is that many home users have asymmetric connection capacity, a situation where ACK compression can occur and cause degradation of TCP throughput [12]. The "emergency mode" has been in the protocol since the first deployment, so we have not been able to evaluate its effects.

A given track can be simultaneously downloaded from the server and several different peers. If a peer is too slow in satisfying a request, the request is resent to another peer or, if getting the data has become too urgent, to the server.

While streaming from a server, clients throttle their requests such that they do not get more than approximately 15 seconds ahead of the current playback point, if there are peers available for the track. When downloading from the peer-to-peer network, no such throttling occurs and the client attempts to download the entire currently playing track. If a user changes tracks, requests relating to the current track are aborted.

Files served within the peer-to-peer network are split into chunks of 16 kB. When determining which peers to request chunks from, the client sorts peers by their expected download times (computed as the number of bytes of outstanding requests from the peer, divided by the average download speed received from that peer) and greedily requests the most urgent chunk from the peer with the lowest estimated download time (and then updates the expected download times). This means that chunks of a track are requested in sequential order. As all peers serving a file have the entire file, requesting blocks in-order does not affect availability or download speeds.

A client can at most have outstanding requests from a given peer for data it believes the peer can deliver within 2 seconds. An exception to this is that it is always allowed to have requests for 32 kB outstanding from a peer. If the estimated download time for a block exceeds the point in time at which the block is needed, that block is not requested.

E. Play-out Delay

Streaming applications need to employ some mechanism to combat the effects of packet loss and packet delay variation. Several different options have been suggested in the literature; for a survey see [13]. Spotify clients do not drop any frames or slow down the playout-rate, and are thus *non delay-preserving* in the nomenclature of [13]. As TCP is used as transport protocol, all data requested will be delivered to the application in-order, but the rate at which data is delivered is significantly affected by network conditions such as packet loss. If a buffer underrun occurs in a track, the Spotify client pauses playback at that point, re-performing latency adjustment.

As discussed by Liang *et al.*[14], there is a trade-off between initial playback latency, receiver buffer size, and the stutter free probability. Spotify clients do not limit the buffer size, and thus the crux of the problem is the appropriate modeling of the channel and using that information to adjust the initial playback latency. As a simplification, the client only considers the channel to the server for latency adjustment.

Spotify clients use a Markovian model for throughput as observed by the client (i.e., affected by packed delay variation, packet loss, and TCP congestion control). Clients make observations of throughput achieved while it is downloading from the server to estimate a Markov chain. Only data collected during the last 15 minutes of downloading is kept and used. The states of the Markov chain is the throughput during 1 second, discretized to 33 distinct levels between 0 and 153 kBps (more granular at lower throughputs).

The model is not used to compute an explicit playback latency. Instead, before playback has commenced, the client periodically uses the Markov chain to simulate the playback of the track, beginning with the current amount of buffered data, and the current data throughput. Each such simulation is considered as failing or passing, depending on if an underrun occurred or not. The client performs 100 simulations and if more than one of them fails, it waits longer before beginning playback. During these simulations the client makes the simplifying assumption that data is consumed at a constant rate despite the fact that the codec used has a variable bitrate encoding.

III. Spotify's Peer-to-Peer Network

Spotify's protocol has been designed to combine server- and peer-to-peer streaming. The primary reason for developing a peer-to-peer based protocol was to improve the scalability of the service by decreasing the load on Spotify's servers and bandwidth resources. An explicit design goal was that the usage of a peer-to-peer network should not decrease the performance in terms of playback latency for music or the amount of stutter. While that design goal is addressed by the reliance on a server for latency-critical parts, it puts demands on the efficiency of the peer-to-peer network in order to achieve good offloading properties.

We discussed in Sections II-B and II-D how the clients combine streaming from the peers and servers. In this section, we give an overview of the peer-to-peer network.

A. General Structure

The peer-to-peer overlay used is an unstructured network, the construction and maintenance of which is assisted by trackers. This allows all peers to participate in the network as equals so there are no "supernodes" performing any special network-maintenance functions. A client will connect to a new peer only when it wishes to download a track it thinks the peer has. It locates peers likely to have a track it is looking for through the mechanisms described in Section III-C.

As discussed in Section II-A, clients store (relatively large) local caches of the tracks they have downloaded. The content of these caches are also what the clients offer to serve to their peers. As tracks are typically small and as live streaming is not supported, a simplification made in the protocol is that a client only offers to serve tracks which it has completely cached. This allows for a slightly simpler protocol, and keeps the protocol overhead down.

There is no general routing performed in the overlay network, so two peers wishing to exchange data must be directly connected. There is a single message forwarded on the behalf of other peers, which is a message searching for peers with a specific track. The rationale for the lack of routing in the overlay is to keep the protocol simple and keep download latencies and overhead down.

B. A Split Overlay Network

The service is currently run from two data centers, one in London and one in Stockholm. A peer uniformly randomly selects which data center to connect to, and load is evenly spread over both data centers. Each data center has an independent peer-to-peer overlay. Thus, the peer-to-peer overlay is in fact split into two overlays, one per site.

The split is not complete since if a client loses its connection to the server, it reconnects to a new server. If it reconnected to the other site it keeps its old peers but is unable to make any new connections to peers connected to servers at its old site. For simplicity of presentation, we will describe the protocol as having a single overlay network.

C. Locating Peers

Two mechanisms are used to locate peers having content the client is interested in. The first uses a tracker deployed in the Spotify back-end, and the second a query in the overlay network.

The problem of locating peers is somewhat different in music-on-demand streaming compared to many other settings.

As tracks are small, a client generally only needs to find one, or a few peers to stream a track from. However, as tracks are also short in duration, downloading new tracks is a very frequent operation, and it is important to minimize the overhead. Furthermore, the lookup time becomes a big issue, which is one of the reasons for Spotify not using a Distributed Hash Table (DHT) to find peers. Other reasons for not implementing a DHT include keeping the protocol simple and keeping overhead down.

The functionality of the tracker is similar, but not identical, to that of a tracker in the BitTorrent protocol [15]. It maintains a mapping from tracks to peers who have recently reported that they have the track. As a peer only offers to serve a track if it has the whole track cached, peers listed in the tracker have the whole track.

As two complementary mechanisms are used, the tracker can be simplified compared to many other system. In particular, the tracker only keeps a list of the 20 most recent peers for each track. Furthermore, clients are only added to the tracker when they play a track and do not periodically report the contents of their caches, or explicitly notify the tracker when content is removed. This helps in keeping overhead down, and simplifies the implementation of the tracker. As clients keep a TCP connection open to a Spotify server, the tracker knows which clients are currently online. When a client asks the trackers for peers who have a track, the tracker replies with up to 10 peers who are currently online. The response is limited in size to minimize overhead.

In addition to the tracker-based peer searches, clients also send search requests in the overlay network, similar to the method used in Gnutella [16]. A client in search of a track sends a search request to all its neighbors in the overlay, who forward the request to all their neighbors. Thus, all peers within distance two of the searcher in the overlay see the request, and send a response back if they have the track cached. Search queries sent by clients have a query id associated with them, and peers remember the 50 most recent searches seen, allowing them to ignore duplicate messages. This limited message forwarding is the only overlay routing in the Spotify peer-to-peer protocol.

When a client is started, how does it get connected to the peer-to-peer network? If it was still listed in the tracker for some tracks then it is possible that other clients will connect to it asking for those tracks. If the user starts streaming a track, it will search the peer-to-peer network and connect to peers who have the track, thus becoming a part of the overlay.

D. Neighbor Selection

Keeping the state required to maintain a large number of TCP connections to peers is expensive, in particular for home routers acting as stateful firewall and Network Address Translation (NAT) devices. Thus, each client has a maximum number of peers it may be connected to at any given time. Clients are configured with both a soft and a hard limit, and never go above the hard limit. The client does not make new connections above the soft limit and periodically prunes its connections to keep itself below the soft limit (with some headroom for new connections). These limits are set to 50 and 60, respectively.

When a client needs to disconnect one or more peers, it performs a heuristic evaluation of the utility of each connected peer. The intention is for the heuristic to take into account both how useful the connection is to the evaluating peer, as well as how useful the link is to the overlay as a whole.

The client sorts all its connected peers according to 6 criteria: bytes sent in the last 10 minutes, bytes sent in the last 60 minutes, bytes received in the last 10 minutes, bytes received in the last 60 minutes, the number of peers found through searches sent over the connection in the last 60 minutes, and the number of tracks the peer has that the client has been interested in in the last 10 minutes. For each criterion, the top scoring peer in that criterion gets a number of points, the second peer a slightly lower number, and so on (with slightly different weights for the different criteria). Peers with a raw score of 0 for a criterion do not get any points for that criterion. The peers points are then summed over all the criteria, and the peers with the least total scores are disconnected.

The client simultaneously uploads to at most 4 peers. This stems from the fact that TCP congestion control gives fairness between TCP connections, so many simultaneous uploads can have adverse effects on other internet usage, in particular for a home user with small uplink bandwidth.

E. State Exchanged Between Peers

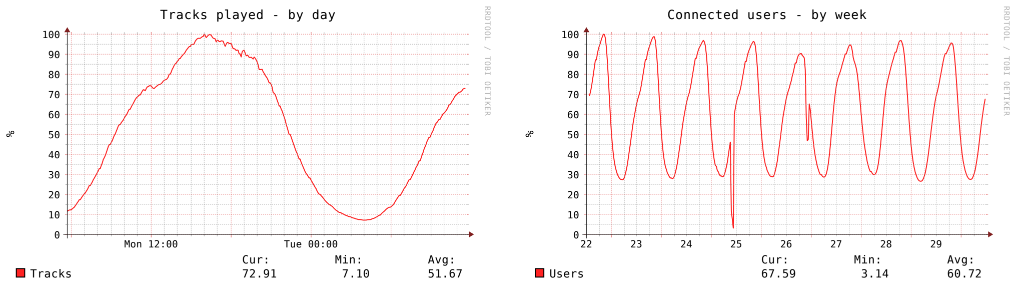
A client wanting to download a track will inform its neighbors of its interest in that track. The interest notification also contains a priority, where the client informs its peer of the urgency of the request. Currently, three discrete levels (in falling order of priority) are used: currently streaming track, prefetching next track, and offline synchronization.

A serving client selects which peers to service requests from by sorting them by the priority of the request, and previously measured upload speed to that peer, and then offer to service the requests of the top 4 peers. Peers are informed whenever their status changes (if their requests become sufficiently prioritized to be serviced, or if their requests no longer are).

F. NAT Traversal

All traffic in the peer-to-peer network uses TCP as transport protocol so the most common protocols for NAT traversal, e.g. STUN [17], are not immediately applicable as they are designed for UDP. While there are techniques for performing TCP NAT traversal as well [18], Spotify clients currently do not perform any NAT traversal.

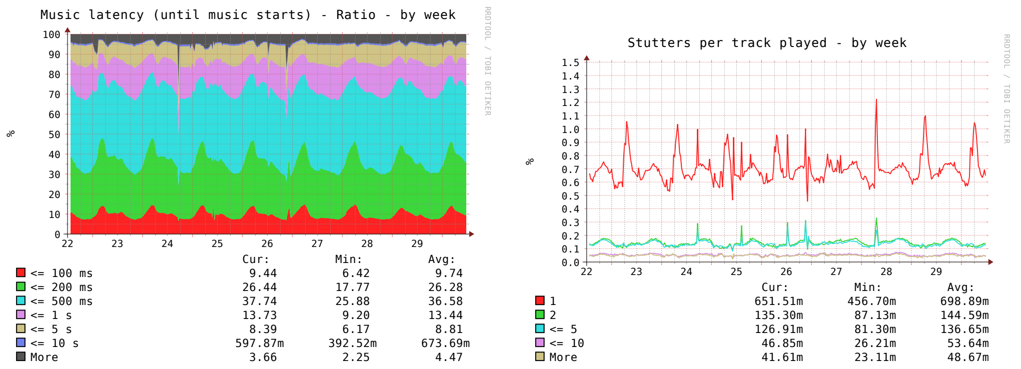
This lack of NAT traversal is mitigated by two factors. Firstly, when a client wishes to connect to a peer a request is also forwarded through the Spotify server asking the connectee to attempt a TCP connection back to the connecter. This allows the connection to be established provided one of the parties can accept incoming connections. Secondly, clients use the Universal Plug n' Play (UPnP) protocol to ask home routers for a port to use for accepting incoming connections.



(a) Tracks played

(b) Users connected

Figure 1. The weekly usage pattern of the Spotify service. Data has been normalized to a 0-1 scale.



(a) Playback latency

(b) Stutter during playback

Figure 3. Playback latency and music stutter over a week.

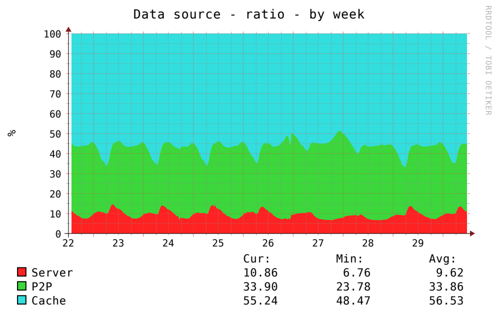


Figure 2. Sources of data used by clients

IV. Peer-to-Peer Evaluation

In this section, we will present and discuss measurements indicating the performance of the Spotify system with focus on the peer-to-peer network performance. For business reasons, some data is presented as ratios rather than absolute volumes.

A. *Measurement Methodology*

Both Spotify clients and servers perform continuous instrumentation and monitoring of the system. Most client measurements are aggregated locally before being sent to the server. For instance, reports on connection statistics are sent every 30 minutes to the server.

The raw log messages are collected and stored on log servers and in a Hadoop cluster (an open-source map-reduce and distributed storage implementation), where they are available for processing. There is also a real-time monitoring system, based on the open-source Munin monitoring system, storing aggregated data and generating graphs based on the log messages and instrumentation of Spotify servers. Most of our graphs are based on the aggregated Munin databases, while most aggregate statistics (e.g., median playback latency) were computed from the raw log files.

In the graphs presented in this paper, min and avg gives the mimimum and average (per time unit) values taken over the measurement period, and cur denotes the current value when the measurement was made. Values below 1 will be denoted with units m or u, denoting milli (10^{-3}) and micro (10^{-6}), respectively.

We collected log messages and monitoring databases for a measurement period of the week between Tuesday 23 March and Monday 29 March 2010 (inclusive). During the measurement period there was a brief disruption (lasting approximately 1.5 hours) in the service during the evening of Friday March 26th. Times are given in UTC.

B. Periodic Variations

When looking at measurements of the Spotify network it quickly becomes apparent that there are significant periodic effects. As users are located in Western Europe, the effect of nighttime is clearly visible. We remark that usage is high throughout the workday, as Spotify is legal and music-listening can be done while working.

There is also a clear weekly pattern with a distinct difference in the shape of data between weekdays and weekends. We believe these effects can likely be attributed to both a difference in user behavior and in the difference of computer setup and network architecture between corporate and home networks. At home, we would expect most users to have machines with much free hard disk space, connected to the Internet through a NAT device speaking UPnP. At work, we would expect a larger variation with many users being behind more restrictive firewalls. In Figure 1, we show the variation over a week in the number of users connected, and the number of tracks played, where the data was normalized to a 0-1 scale. As the two curves have almost the same shape we use different time frames for the two graphs, and show tracks played during a single day in Figure 1(a), and users connected over a week in Figure 1(b). The dip during the early morning hours of March 25th appears to be due to an error in the collection of the data, while the dip during the 26th is due to the outage mentioned above.

C. Data Sources

We now turn to the question of how effectively the servers are offloaded. Figure 2 shows the sources of track data for the client and its variation throughout a week. All (non-duplicate) data downloaded from servers and peers is counted, even if it is not played (due to the user skipping the track). Data from the cache is counted when the client reads the data from its cache for playback, meaning that it corresponds reasonably closely, but not exactly, to cached data played by the client.

Some periodic effects are clearly visible. During nighttime, a significantly larger fraction of data played comes from the cache. This effect is less pronounced on weekend nights, when users play more new music. There is a distinct decrease in the fraction offloaded by the peer-to-peer overlay while the logged in population is increasing during the morning, and there is a lesser, but more prolonged decrease when users log off in the evening. There is also a difference between weekdays and weekends, with peer-to-peer data offloading a larger fraction of data during the weekends.

In total, during the measurement period, 8.8% of data came from servers, 35.8% from the peer-to-peer network, and the remaining 55.4% were cached data. Thus, we can see that the large caches and peer-to-peer network together significantly decrease the load on Spotify's servers. Our results are comparable to the measurements performed by Huang *et al.* [5] where PPLive was shown to have 8.3% data downloaded from servers.

D. Playback Latency and Stutter

Two important measures for on demand streaming services are the playback latency and the stutter frequency. As discussed by Liang *et al.* [14], streaming applications must make a trade-off between the two quantities. We are not aware of any studies discussing in detail how user's satisfaction with a streaming service depends on these two factors, but we believe them to be very important. We think that studying the impact of streaming performance on user satisfaction would be an interesting subject for future research.

When combining peer-to-peer and server streaming, there is also a trade-off between server load and latency and stutter frequencies. Spotify has designed to prioritize the latter two.

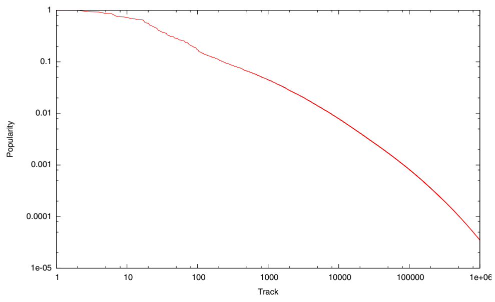
The Spotify client measures the playback latency as the time between when a track should start, either due to user action or due to the previous track ending, and the time the OS is instructed to begin playback. Thus, the latency measured not only includes network latency but also the time needed on the client's host to decrypt and decompress downloaded data. We remark that, due to digital rights management (DRM), even fully cached material must wait for a reply over the network before playback can commence and such a network request is sent when the track should start, unless the track has been synchronized for offline playback. Thus, the latency almost always includes at least one RTT to the Spotify servers.

In Figure 3(a), we show how playback latency varied over the measurement period. The large fraction of playbacks shown as having a latency of more than 10 seconds was discovered to be due to an error in latency reporting, the true value is significantly lower, and was below 1% throughout the measurement period.

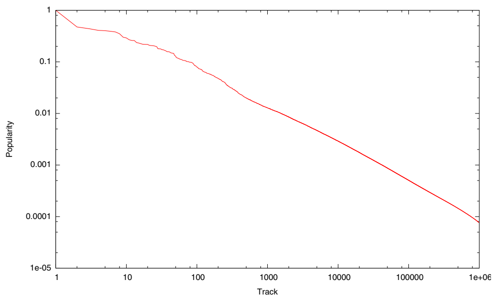
We observe that the latency is significantly lower during the night, when the number of tracks played is also low. This is mainly due to two reasons. Firstly, more data is played from the cache during the night. Secondly, users are less interactive during the night, allowing the prefetching discussed in Section II-B to be much more efficient.

Throughout the measurement period the median latency was 265 ms, the 75th percentile was 515 ms, and the 90th percentile was 1047 ms. As over half the data during the measurement period came from the cache, it is likely that the median playback latency measures the time needed for DRM together with local processing time.

Intimately related to the playback latency is the amount of stutter occurring upon playback. In Figure 3(b), we show the fraction of playbacks during which one or more stutters occurred. During the measurement period, less than 1% of all playbacks stuttered. However, in 64% of stutters, the client had more than 20 kilobytes of compressed music data available. This indicates that stutters are often due to local effects (e.g.,

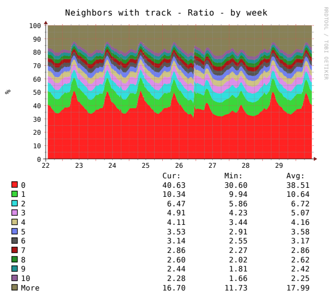


(a) Track playback frequencies (normalized), log-log scale



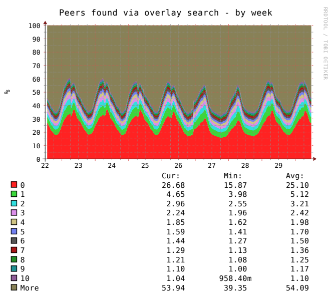
(b) Track server request frequencies (normalized), log-log scale

Figure 4. Frequency of track accesses, both as played by clients and as blocks requested from the Spotify servers



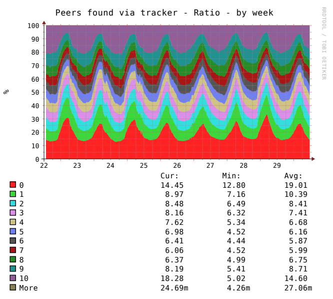
	Cur:	Min:	Avg:
🟥 0	40.63	30.60	38.51
🟩 1	10.34	9.94	10.64
🟦 2	6.47	5.86	6.72
🟪 3	4.91	4.23	5.07
🟨 4	4.11	3.44	4.16
🟦 5	3.53	2.91	3.58
⬛ 6	3.14	2.55	3.17
🟥 7	2.86	2.27	2.86
🟩 8	2.60	2.02	2.62
🟦 9	2.44	1.81	2.42
🟪 10	2.28	1.66	2.25
🟫 More	16.70	11.73	17.99

(a) Neighbors who had track



	Cur:	Min:	Avg:
🟥 0	26.68	15.87	25.10
🟩 1	4.65	3.98	5.12
🟦 2	2.96	2.55	3.21
🟪 3	2.24	1.96	2.42
🟨 4	1.85	1.62	1.98
🟦 5	1.59	1.41	1.70
⬛ 6	1.44	1.27	1.50
🟥 7	1.29	1.13	1.36
🟩 8	1.21	1.08	1.25
🟦 9	1.10	1.00	1.17
🟪 10	1.04	958.40m	1.10
🟫 More	53.94	39.35	54.09

(b) Peers with track found in overlay



	Cur:	Min:	Avg:
🟥 0	14.45	12.80	19.01
🟩 1	8.97	7.16	10.39
🟦 2	8.48	6.49	8.41
🟪 3	8.16	6.32	7.41
🟨 4	7.62	5.34	6.68
🟦 5	6.98	4.52	6.16
⬛ 6	6.41	4.44	5.87
🟥 7	6.06	4.52	5.99
🟩 8	6.37	4.99	6.75
🟦 9	8.19	5.41	8.71
🟪 10	18.28	5.02	14.60
🟫 More	24.69m	4.26m	27.06m

(c) Peers returned by tracker

Figure 5. Peers found through broadcasting searches in the overlay and through the tracker

not receiving sufficient CPU time), rather than the network. While we think a rate of 1% is acceptably low, there seems to be room for further improvement.

E. Distribution of Track Accesses

There has been a considerable amount of discussion on how an on-demand system affects usage patterns [19], [20], [21]. In particular, it has been discussed whether all-you-can-eat on-demand access leads users to access content according to a long-tail distribution. A full discussion of the access pattern of music in the Spotify service is out of scope for this paper, but we will briefly mention some findings on the access pattern.

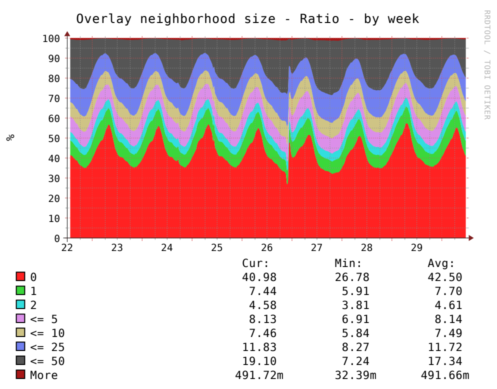
In Figure 4(a), we show the shape of the probability density function of track playbacks in the Spotify system, normalized so the most popular item has frequency 1. During the measurement period 88% of the track accesses were within the most popular 12% of the library. While much weight is on the most popular tracks, it turns out that a large fraction of the catalog is played. During our week-long measurement period, approximately 60% of the content available was accessed at least once.

Given the distribution of track accesses one could wonder how well the peer-to-peer network works at varying degrees of popularity. In Figure 4(b), we show the shape of frequencies of tracks as requested from the Spotify servers. The graph is based on data collected during February 21–23 2010, which is outside our measurement period. This graph gives the relative frequencies of tracks requested at servers. Clients normally make between zero and ten server requests for a track, depending on how much of it they download from a server. The size of blocks requested varies, with the mean size being 440 kB. The distribution of requests to the servers is significantly less top-heavy than that of track playbacks with 79% of accesses being within the most popular 21%.

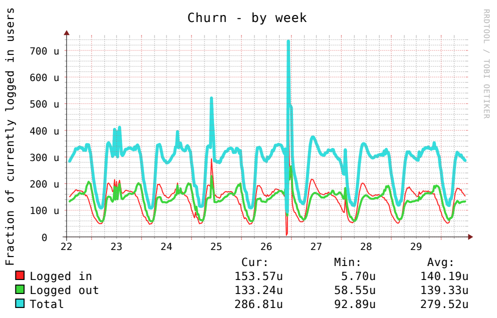
F. Locating Peers

Spotify provides two separate mechanisms for locating peers having a piece of content, as described in Section III-C. In Figure 5, we show diagrams indicating how well the mechanisms work. We have divided the neighborhood search into two types of responses, peers who were already neighbors (Figure 5(a)), and new peers found who were at distance 2 (Figure 5(b)). We observe that locating peers through the overlay appears to be essentially binary in nature—most requests receive either none or many replies. In contrast, the number of peers located through the tracker mechanism is more varied.

The fact that a client so often finds that many of its overlay neighbors have the track it is interested in indicates that a certain amount of clustering by interest is occurring in the overlay. We believe it would be an interesting topic for future



(a) Neighborhood size



(b) Churn

Figure 6. Overlay properties — sizes of peer's neighborhoods, and churn (fraction of currently connected clients connecting/disconnecting per second)

Table I
SOURCES OF PEERS

Sources for peers	Fraction of searches
Tracker and P2P	75.1%
Only Tracker	9.0%
Only P2P	7.0%
No peers found	8.9%

Table II
DISTRIBUTION OF APPLICATION-LAYER TRAFFIC IN OVERLAY NETWORK

Type	Fraction
Music data, used	94.80%
Music data, unused	2.38%
Search overhead	2.33%
Other overhead	0.48%

work to further explore the extent of such clustering, and if it can be further enhanced by the protocol design, or if it can be leveraged in other peer-to-peer systems.

Recall that the tracker mechanism is necessary for bootstrapping a newly joined client into the overlay network (as without a tracker, she would never get connected to anyone and could not use searches in the overlay to find new peers). Referring to Figure 5(c), it seems as if once the bootstrapping was done it is possible that the searching within the overlay could be a sufficiently effective mechanism on its own.

When designing the two complementary mechanisms, a reason for having both mechanisms was that the tracker would allow for "long jumps" within the overlay. Assuming that some amount of clustering is occurring, if a user's mood changes and she now feels like listening to Jazz instead of the Hip Hop she has been listening to for the last hour, intuitively it seems that searching locally in her overlay would not yield much, if anything. A reason for having overlay searches given that there is a tracker is to make the system more robust to tracker failures, and to allow for a simpler and cheaper tracker implementation.

We analyzed the traces for those tracks where the client had downloaded at least one byte. Thus, tracks played from the cache, or skipped before the client received any data were ignored in this analysis. The results of a tracker query and a neighborhood search can overlap, and a client may learn of a given peer from both sources simultaneously. Such peers are reported by the client as having been found in the peer-to-peer network (only).

The results are summarized in Table I. We note that, for a majority of requests, both mechanisms are successful in locating new peers. We believe that the relatively high frequency (7%) of the event that peers are found only through the overlay is mostly an artifact of the reporting of overlapping search results.

G. Peer-to-Peer Protocol Overhead

An issue in all peer-to-peer networks is to keep the overhead cost of the overlay at a minimum. A potential problem in Gnutella-like systems in particular is the overhead incurred by searches in the peer-to-peer network.

Clients periodically report the total number of bytes received on peer-to-peer TCP sockets as well as the total number of bytes of useful song data they have downloaded over the peer-to-peer network. Song data is considered useful if the request had not been cancelled before the data arrived, and if it is not duplicate to data the client has received from other peers or the server. Our measurements do not include overhead caused by lower-layer protocols such as TCP and IP.

Over our measurement period, 5.20% of the traffic received from the peer-to-peer network was not useful song data. In Table II we break down the overhead into three categories. Most of the overhead, 2.38% of received traffic, comes form song data downloaded which was not useful to the client. Closely following is the overhead coming from searching for peers. The measurements on protocol overhead can be compared with experimental measurements on the BitTorrent protocol by Legout *et al.* [7] where they found the BitTorrent protocol overhead to be, in most of their measurements, below 2%. We remark that the two protocols are quite different in nature, and there is a difference in the measurement methodology.

H. Properties of the Overlay Network

As discussed in Section III-D, each peer has a limit to the number of neighbors it retains connections to in the overlay network. In Figure 6(a), we show what the distribution of node degrees in the overlay looks like. A surprisingly large fraction (between 30%-50%) of clients are disconnected from the overlay. While we have not fully investigated the reasons for this, we believe a partial explanation may be that the neighbor selection and tracker mechanisms cause idle users to become disconnected from the overlay.

In Section III-F, we discussed the absence of NAT traversal techniques in the Spotify clients. Intuitively, one would expect that this would constitute a problem as users are likely to be situated behind various NAT devices. Our data shows that it indeed is a problem and that the fraction of successful connection attempts was 35% during our measurement period.

An important factor for how well a peer-to-peer network can function is the amount of churn. While we do not have any direct measurements of churn, we do have access to data showing the fraction of currently connected clients connecting to and disconnecting from servers, shown in Figure 6(b). It can happen that a client reconnects to a new Spotify server while still being active in the peer-to-peer network, e.g. in case the server is overloaded. This should be sufficiently rare that Figure 6(b) gives a realistic measurement of the churn in the overlay.

We note that the total churn rate is roughly even throughout the day with a sharp decrease during nighttime. During the morning it is dominated by logins while it is dominated by logouts towards the evening. Comparing the churn to Figure 2 we observe that, seemingly, the efficiency of the overlay in data delivery is not severely impacted by clients logging out. On the other hand, there is a daily dip in the fraction of data delivered by the overlay during the morning when many new users are logging on.

V. Conclusion

We have given an overview of the protocol and structure of the Spotify on-demand music streaming service, together with many measurements of the performance of the system. In particular, we note that the approach that Spotify uses to combine server-based and peer-to-peer streaming gives very good results, both with respect to user-relevant performance measures, and in reducing server costs. Furthermore, this is done using TCP as a transport protocol, indicating that streaming over TCP is a very viable option. The data collected shows also shows that a simplified tracker coupled with overlay broadcasts can be an efficient design for locating peers.

We believe on-demand streaming will continue to grow rapidly in the coming years, and that many interesting problems remain in further developing such services. Among these are (1) development of user satisfaction measures for on-demand streaming; (2) improved playout strategies, adapted to peer-to-peer data delivery; (3) efficient peer-to-peer overlays exploiting the overlap in interests between users.

Acknowledgements

We would like to express our gratitude to everyone working at Spotify for their help. We would like to thank Sonja Buchegger, Pehr Söderman, and Cenny Wenner for reading and commenting on draft versions.

References

[1] N. Magharei, R. Rejaie, and Y. Guo, "Mesh or multiple-tree: A comparative study of live P2P streaming approaches," in *Proc. of IEEE INFOCOM'07*, 2007, pp. 1424–1432.

[2] H. Deshpande, M. Bawa, and H. Garcia-Molina, "Streaming live media over peers," Stanford InfoLab, Technical Report 2002-21, 2002. [Online]. Available: http://ilpubs.stanford.edu:8090/863/

[3] X. Zhang, J. Liu, B. Li, and T.-S. P. Yum, "CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming," in *Proc. of IEEE INFOCOM'05*, 2005, pp. 2102 – 2111.

[4] X. Liao, H. Jin, Y. Liu, L. M. Ni, and D. Deng, "AnySee: Peer-to-peer live streaming," in *Proc. of IEEE INFOCOM'06*, 2006, pp. 1–10.

[5] Y. Huang, T. Z. Fu, D.-M. Chiu, J. C. Lui, and C. Huang, "Challenges, design and analysis of a large-scale P2P-VoD system," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 375–388, 2008.

[6] X. Hei, C. Liang, J. Liang, Y. Liu, and K. Ross, "A measurement study of a large-scale P2P IPTV system," *Multimedia, IEEE Transactions on*, vol. 9, no. 8, pp. 1672–1687, dec. 2007.

[7] A. Legout, G. Urvoy Keller, and P. Michiardi, "Understanding BitTorrent: An experimental perspective," Technical Report, 2005. [Online]. Available: http://hal.inria.fr/inria-00000156/en/

[8] S. Saroiu, K. P. Gummadi, and S. D. Gribble, "A measurement study of peer-to-peer file sharing systems," in *Multimedia Computing and Networking (MMCN)*, January 2002.

[9] S. Floyd, T. Henderson, and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC 3782 (Proposed Standard), Internet Engineering Task Force, Apr. 2004.

[10] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, 2008.

[11] M. Allman, V. Paxson, and E. Blanton, "TCP Congestion Control," RFC 5681 (Draft Standard), Internet Engineering Task Force, Sep. 2009.

[12] L. Kalampoukas, A. Varma, and K. K. Ramakrishnan, "Improving TCP throughput over two-way asymmetric links: analysis and solutions," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, pp. 78–89, 1998.

[13] N. Laoutaris and I. Stavrakakis, "Intrastream synchronization for continuous media streams: a survey of playout schedulers," *Network, IEEE*, vol. 16, no. 3, pp. 30–40, may/jun 2002.

[14] G. Liang and B. Liang, "Jitter-free probability bounds for video streaming over random VBR channel," in *Proc. of QShine'06*. New York, NY, USA: ACM, 2006, p. 6.

[15] B. Cohen, "The bittorrent protocol specification," BEP 3 (Standard), 2008. [Online]. Available: http://www.bittorrent.org/beps/bep_0003.html

[16] M. Ripeanu, "Peer-to-peer architecture case study: Gnutella network," *Proc. of IEEE P2P'01*, pp. 99–100, 2001.

[17] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session Traversal Utilities for NAT (STUN)," RFC 5389 (Proposed Standard), Internet Engineering Task Force, Oct. 2008.

[18] S. Perreault and J. Rosenberg, "TCP candidates with interactive connectivity establishment (ICE)," Internet-Draft (work in progress), draft-ietf-mmusic-ice-tcp-08, Oct. 2009. [Online]. Available: http://tools.ietf.org/html/draft-ietf-mmusic-ice-tcp-08

[19] C. Anderson, *The Long Tail: Why the Future of Business Is Selling Less of More*. Hyperion, July 2006. [Online]. Available: http://www.worldcat.org/isbn/1401302378

[20] A. Elberse, "Should you invest in the long tail?" *Harvard Business Review*, vol. 86, no. 7/8, pp. 88–96, 2008.

[21] S. Goel, A. Broder, E. Gabrilovich, and B. Pang, "Anatomy of the long tail: ordinary people with extraordinary tastes," in *WSDM '10: Proceedings of the third ACM international conference on Web search and data mining*. New York, NY, USA: ACM, 2010, pp. 201–210.

II

A Zero-One Law for Secure Multi-Party Computation with Ternary Outputs (full version)

Gunnar Kreitz

KTH – Royal Institute of Technology
gkreitz@kth.se

Abstract. There are protocols to privately evaluate any function in the passive (honest-but-curious) setting assuming that the honest nodes are in majority. For some specific functions, protocols are known which remain secure even without an honest majority. The seminal work by Chor and Kushilevitz [7] gave a complete characterization of Boolean functions, showing that each Boolean function either requires an honest majority, or is such that it can be privately evaluated regardless of the number of colluding nodes.
The problem of discovering the threshold for secure evaluation of more general functions remains an open problem. Towards a resolution, we provide a complete characterization of the security threshold for functions with three different outputs. Surprisingly, the zero-one law for Boolean functions extends to $\mathbb{Z}_3$, meaning that each function with range $\mathbb{Z}_3$ either requires honest majority or tolerates up to n colluding nodes.

1 Introduction

Multi-party secure function evaluation (SFE) is a cornerstone of modern cryptography, and has been extensively studied since it was introduced by Yao [14]. In this work we consider the joint evaluation by n parties of a public n-ary function f in such a way that no collusion of parties learns anything more than what they do by knowing their own inputs and seeing the output. We consider the *symmetric* case where all participants receive the same output.

Several models of adversaries occur in the SFE literature. A first distinction is whether the adversary has limited computational power (*computational security*) or not (*information-theoretic security*). A second important distinction is whether the parties corrupted by the adversary must still follow the protocol (*passive*) or not (*active*). In the present work, we are concerned with information-theoretic security and all adversaries considered are passive. We assume that the parties communicate over a complete network with *private channels*, meaning that the adversary cannot see messages sent between two honest parties.

Another important limitation put upon the adversary is which parties she can corrupt. The most common adversary is allowed to corrupt up to a threshold $t \leq n$ participants for some t which is typically a function of n. We say that a function for which there is a protocol tolerating up to t corruptions is t-private. In this paper, we will only consider threshold adversaries. More general adversarial models have also been studied, both in terms of a more general specification of the parties the adversary can corrupt by Hirt and Maurer [10] and considering a mix active and passive adversarial corruptions by Beerliová-Trubíniová *et al.* [2].

There exist protocols to securely evaluate any function $\lfloor (n-1)/2 \rfloor$-privately in our setting by Ben-Or, Goldwasser, and Wigderson [3], and Chaum, Crépeau, and Damgård [4]. For some functions, in particular Boolean disjunction, this has been proved to be an upper bound meaning that there are no protocols to evaluate them which remain secure against more than $\lfloor (n-1)/2 \rfloor$ colluding parties. For other functions, in particular summation over a finite Abelian group, there are n-private protocols. This raises the question of determining the privacy threshold of functions.

Chor and Kushilevitz [7] completely answered the question for Boolean functions. They proved a zero-one law showing that each Boolean function is either $\lfloor (n-1)/2 \rfloor$-private (and not $\lceil n/2 \rceil$-private) or n-private. Their work presents a proof that a function containing an OR-like substructure (an *embedded* OR) is $\lfloor (n-1)/2 \rfloor$-private and that all Boolean functions without such a substructure can be computed by a single Boolean summation.

Proving that a function f cannot be t-privately computed is often done by a partition argument, reducing to the two-party case. In these proofs, the parties are partitioned into two parts of size $\leq t$ and we think of f as a two-party function with each party supplying all inputs for one set of the partition. If the two-party function is not 1-private, then f is not t-private. Chor and Ishai [6] analyzed partition arguments and gave a generalization partitioning the parties into $k > 2$ sets which increases the power of the framework. However, in this paper, we will only need partitioning arguments with two sets.

Chor, Geréb-Graus, and Kushilevitz [5] showed that for every t, $\lceil n/2 \rceil \leq t \leq n-2$ there exists a function such that it is t-private but not $(t+1)$-private. We remark that the functions they construct in their proofs have very large ranges which grow exponentially with t.

The privacy of symmetric[1] functions with Boolean arguments has been studied by Chor and Shani [9]. For such functions, they prove a necessary condition on the preimages of outputs for the function to be $\lceil n/2 \rceil$-private. They also define a class called dense symmetric functions where this necessary condition is also sufficient for n-privacy. Thus, they also prove a zero-one law where for a class of functions, where each function in the class is either n-private or not $\lceil n/2 \rceil$-private.

For two-party computation, a complete characterization of the 1-private functions was made independently by Beaver [1] and Kushilevitz [13]. They both show that a function f is 1-private if and only if it is decomposable, and for decomposable functions, there is a straightforward 1-private protocol. One of

[1] Here, symmetric means the standard notion of a symmetric function, not the SFE-specific notion that all parties receive the same output.

our protocols, Protocol 3, can be viewed as a generalization of the protocol for decomposable functions to the multi-party case.

Künzler, Müller-Quade, and Raub [12] give a combinatorial classification of functions computable in several different adversarial models, including the information-theoretic passive model which we work with in this paper. However, in this setting, they consider the broadcast model of communication which gives different results from private channels. For instance, summation is not n-private in the broadcast channel model.

1.1 Our Contribution

In this work, we extend the zero-one law of Boolean privacy to functions with three outputs. For notational convenience, we talk about functions with range $\mathbb{Z}_3$, but we would like to emphasize our results do not depend on any algebraic structure over the range of the function. More formally, we prove the following statement:

Theorem 1 (Main theorem). *For every n-argument function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$, f is either n-private, or it is $\lfloor (n-1)/2 \rfloor$-private and not $\lceil n/2 \rceil$-private.*

The core part of our proof is a structure lemma (Lemma 8) showing that every function f with range $\mathbb{Z}_3$ must have at least one of three properties (which we define more formally later):

- f has an embedded OR
- f is a permuted sum
- f is collapsible.

We provide protocols for n-privately evaluating those functions of the two latter types which do not contain an embedded OR.

Our definition of an embedded OR is a generalization of the one commonly found in the literature, but the presence of one implies that there is no protocol which can securely evaluate f and tolerate more than t colluding parties for some t (but potentially for a $t > \lceil n/2 \rceil$).

Finally, we prove (Theorem 22) that the existence of an embedded OR (in our generalized sense) also implies the existence of a "small" embedded OR, giving $t = \lceil n/2 \rceil$. By combining this result with our structure lemma and the result from [7] that a function with an embedded OR of size at most $\lceil n/2 \rceil$ cannot be $\lceil n/2 \rceil$-privately computed, our main theorem follows. We state the proof more formally in Section 6.

We remark that while our statements are true for $n = 2$, there are complete classifications [1,13] for the 2-party case which are simpler than ours (for $n = 2$, our protocols reduce to decomposition) and not limited to functions with range $\mathbb{Z}_3$. Our contribution lies in the case when $n \geq 3$.

The proof of our theorems are significantly more involved than the analogous proofs for Boolean functions. In several of our proofs we need to apply a fairly extensive case analysis.

Our result answers in part a question raised by Chor and Ishai [6] by showing that partition reductions (with only two sets) are universal for proving non-privacy of functions mapping to $\mathbb{Z}_3$.

2 Notation and Preliminary Theorems

We use boldface letters to refer to vectors, like: $\boldsymbol{x}$, $\boldsymbol{y}$. We work with functions with range $\mathbb{Z}_3$, and use the three Greek letters α, β, and γ to denote the three different outputs of the function. We take as convention that the three represent distinct outputs (so $\alpha \neq \beta \neq \gamma$). Sometimes we need to discuss an output as being not α, which we denote by $\not\alpha$.

In the proceeding discussion, we often need to discuss the behavior of a subfunction when keeping some subset of its arguments fixed. To simplify this discussion, we introduce some notation. For disjoint $S_1, S_2, S_3 \subseteq [n]$ we define

$$\begin{aligned} f^{\boldsymbol{a}}_{\{S_1\}}(\boldsymbol{x}) &\stackrel{\text{def}}{=} f(\{x_i\}_{i\in S_1}, \{a_i\}_{i\in S_1^C}) \\ f^{\boldsymbol{a}}_{\{S_1,S_2\}}(\boldsymbol{x},\boldsymbol{y}) &\stackrel{\text{def}}{=} f(\{x_i\}_{i\in S_1}, \{y_i\}_{i\in S_2}, \{a_i\}_{i\in (S_1\cup S_2)^C}) \\ f^{\boldsymbol{a}}_{\{S_1,S_2,S_3\}}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) &\stackrel{\text{def}}{=} f(\{x_i\}_{i\in S_1}, \{y_i\}_{i\in S_2}, \{z_i\}_{i\in S_3}, \{a_i\}_{i\in (S_1\cup S_2\cup S_3)^C}) \,. \end{aligned}$$

We sometimes consider singleton sets S_1, S_2, S_3 and then denote them simply by their only element, with some abuse of notation. That is,

$$\begin{aligned} f^{\boldsymbol{a}}_{\{i\}}(x) &\stackrel{\text{def}}{=} f(a_1,\ldots,a_{i-1},x,a_{i+1},\ldots,a_n) \\ f^{\boldsymbol{a}}_{\{i,j\}}(x,y) &\stackrel{\text{def}}{=} f(a_1,\ldots,a_{i-1},x,a_{i+1},\ldots,a_{j-1},y,a_{j+1},\ldots,a_n)\,, \end{aligned}$$

and analogously for $f^{\boldsymbol{a}}_{\{i,j,k\}}(x,y,z)$ and $f^{\boldsymbol{a}}_{\{i,j,k,l\}}(x,y,z,w)$.

We need to describe details of functions' behaviors, and adopt a geometric viewpoint. In the proofs, we speak of inputs as being neighbors and of rows, diagonals, and rectangles and induced rectangles in the function table. By neighbors we mean points at Hamming distance 1. By a row, we mean the values taken by the function fixing all but one values, i.e. the values $f^{\boldsymbol{a}}_{\{i\}}(x)$ for all $x \in A_1$ with a fixed i and $\boldsymbol{a}$ which are clear from the context. By a rectangle, we mean the values $f^{\boldsymbol{e}}_{\{S_1,S_2\}}(\boldsymbol{a},\boldsymbol{c}), f^{\boldsymbol{e}}_{\{S_1,S_2\}}(\boldsymbol{a},\boldsymbol{d}), f^{\boldsymbol{e}}_{\{S_1,S_2\}}(\boldsymbol{b},\boldsymbol{c}), f^{\boldsymbol{e}}_{\{S_1,S_2\}}(\boldsymbol{b},\boldsymbol{d})$. Note that a rectangle by this definition is a high-dimensional structure. By induced rectangle, we mean a rectangle as before but where $|S_1| = |S_2| = 1$, thus looking like a rectangle in the function table. We only use the concept of a diagonal of a 2×2 induced rectangle. For fixed inputs $\boldsymbol{a}$ and dimensions i, j we say that $f^{\boldsymbol{a}}_{\{i,j\}}(x_1,y_1), f^{\boldsymbol{a}}_{\{i,j\}}(x_2,y_2)$ is a diagonal for $x_1 \neq x_2$ and $y_1 \neq y_2$.

Definition 1 (Redundant inputs). *For an n-argument function f, we say that inputs $x, y, x \neq y$ are* redundant for player k *if for all $\boldsymbol{a}$ it holds that $f^{\boldsymbol{a}}_{\{k\}}(x) = f^{\boldsymbol{a}}_{\{k\}}(y)$.*

Definition 2 (Normalized function). *An n-argument function f with no redundant inputs for any player is said to be* normalized.

We take as convention that all functions are normalized. This assumption is without loss of generality as a function can easily be normalized by for each set of redundant inputs removing all but one. A protocol for evaluating the normalized function can be used to evaluate the original function as well by performing the same procedure.

To prove Theorem 1, we make use of a theorem by Chor and Kushilevitz [7] which states that there is no 1-private protocol for a 2-party computation of disjunction. Through standard simulation techniques, this gives impossibility results for multi-party protocols of functions containing an OR-like substructure. This is commonly referred to as an embedded OR, or a corner. We formally define an embedded OR and then restate their result. For a two-party function, the definition is straightforward:

Definition 3 (Embedded OR (2 parties)). *We say that a two-argument function f contains an* embedded OR *if there exists inputs x_1, x_2, y_1, y_2 ($x_1 \neq x_2, y_1 \neq y_2$) such that $f(x_1, y_1) = f(x_1, y_2) = f(x_2, y_1) \neq f(x_2, y_2)$.*

However, when considering the n-party case, the definition of an embedded OR becomes slightly more complex. In particular, we need our definition to capture the size of the collusion required to realize an embedded OR, as that size also limits the impossibility result that follows from the existence of such an embedded OR. To this end, we define an embedded OR as having a degree k. We remark that Kilian *et al.* [11] define an embedded OR as one of degree 1. Much of the previous literature has mostly been concerned with Boolean functions, and then, the existence of an embedded OR (of any degree) implies the existence of one of degree 1, as proved in [11]. However, for functions with larger ranges, the situation is more complex, as shown by our Theorem 22.

Definition 4 (Embedded OR (n parties, induced, generalized), corner-free). *We say that an n-argument function f contains an* embedded OR *of degree k if there exists disjoint subsets $S_1, S_2 \subset [n]$ where $|S_1|, |S_2| \leq k$, and values $\boldsymbol{a}$ such that the two-argument function $f'(\boldsymbol{x}, \boldsymbol{y}) = f^{\boldsymbol{a}}_{\{S_1,S_2\}}(\boldsymbol{x}, \boldsymbol{y})$ contains an embedded OR. We refer to an embedded OR of degree 1 as an* induced *embedded OR, and one of degree greater than 1 as a* generalized *embedded OR. A function without an embedded OR (of any degree) is said to be* corner-free.

With the definitions in place, we are ready to restate a result by Chor and Kushilevitz [7]. The result we need was not presented as a separate lemma in their paper, but instead follows as a corollary from two of their lemmas which we restate in simplified form.

Lemma 2 (Partition lemma, [7]). *Let $f : A_1 \times \ldots \times A_n \to R$ be $\lceil n/2 \rceil$-private. Then for every subset S_1 of size $\lceil n/2 \rceil$, the two-argument function $f'(\boldsymbol{x}, \boldsymbol{y}) = f_{\{S_1, S_1^C\}}(\boldsymbol{x}, \boldsymbol{y})$ is 1-private.*

Lemma 3 (Corners lemma, [7]). *A two-argument function is not 1-private if it contains an embedded* OR.

Corollary 4. *A function containing an embedded* OR *of degree at most* $\lceil n/2 \rceil$ *is not* $\lceil n/2 \rceil$*-private.*

We also make use of [7, Theorem 4] which states that a corner-free Boolean function can be expressed as a Boolean sum:

Theorem 5 ([7]). *For a corner-free Boolean function* f *there are functions* f_i *such that* $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} f_i(x_i)$ *where the sum is computed modulo 2.*

We formally restate the theorem from [11] showing that a generalized embedded OR in a Boolean function implies an induced embedded OR. In our terminology:

Theorem 6 ([11]). *A Boolean function* f *containing an embedded* OR *contains an embedded* OR *of degree 1.*

We show functions and subfunctions which depend on up to 4 arguments. To be able to draw them, we show 2-dimensional projections separated by lines with vertical lines indicating a 3^{rd} dimension and horizontal lines indicating a 4^{th} dimension. We present sample function in Figure 1, showing a function which contains an embedded OR of degree 2 but does not contain an embedded OR of degree 1. The highlighted embedded OR occurs with the subsets $S_1 = \{P_1, P_3\}$ and $S_2 = \{P_2, P_4\}$ with inputs $(2, 1)$ and $(1, 2)$ for S_1 and $(1, 1)$ and $(2, 2)$ for S_2. As the function is drawn, the coalition S_1 in the embedded OR controls the horizontal position, and S_2 controls the vertical position.

0	**1**	**1**	2
1	0	2	1
2	0	0	1
0	**2**	**1**	0

Fig. 1. An example function containing an embedded OR of degree 2 (highlighted).

We use the following lemma which we believe is well-known. For completeness, we include a proof in the appendix.

Lemma 7. *If an n-argument function* $f : A_1 \times \ldots \times A_n \to G$*, where* G *is an Abelian group, has the property that for every pair of dimensions* j, k *and inputs* $x_1, x_2, y_1, y_2, \boldsymbol{a}$ *the following equality holds:*

$$f^{\boldsymbol{a}}_{\{j,k\}}(x_1, y_1) + f^{\boldsymbol{a}}_{\{j,k\}}(x_2, y_2) = f^{\boldsymbol{a}}_{\{j,k\}}(x_1, y_2) + f^{\boldsymbol{a}}_{\{j,k\}}(x_2, y_1), \tag{1}$$

then f *can be rewritten as* $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} f_i(x_i)$*.*

3 A Structure Lemma

The main step towards proving Theorem 1 is the establishment of a structure lemma for functions with range $\mathbb{Z}_3$. Thus, we turn toward some global properties of functions (as opposed to the comparatively local property of the existence of an embedded OR). The first such property captures the case when we can split the range of a function into two parts, and compute a Boolean sum to discover which part the output lies in. If we can then proceed with further such subdivisions until we arrive at a single possible output, this immediately gives a protocol to compute f. We prove that this further subdivision is always possible for corner-free f with range $\mathbb{Z}_3$ in Lemma 21. We remark that this is a further generalization of the multi-party decomposability defined in [12], which in turn was a generalization of 2-party decomposability defined in [13]. We show a collapsible function and the generalized decomposition of it in Figure 2.

Definition 5 (Collapsible). *We say that a function $f : A_1 \times \ldots A_n \to R$ is* collapsible *if there is a subset $R', \emptyset \subset R' \subset R$ such that the Boolean function*

$$f'(\boldsymbol{x}) = \begin{cases} 1 & \textit{if } f(\boldsymbol{x}) \in R' \\ 0 & \textit{otherwise} \end{cases}$$

does not contain an embedded OR *and can thus be n-privately computed. We refer to f' as being collapsed.*

For a collapsible function f with range $\mathbb{Z}_3$ if f is collapsible we can choose R' with two elements α, β and say that f is collapsible by collapsing α and β.

$$\begin{array}{ccc|ccc} 0&1&2&2&2&0 \\ 1&0&2&2&2&1 \\ 2&2&0&1&0&2 \end{array} \qquad\qquad \begin{array}{ccc|ccc} 1&1&0&0&0&1 \\ 1&1&0&0&0&1 \\ 0&0&1&1&1&0 \end{array}$$

(a) Collapsible f (b) f collapsed

Fig. 2. An example collapsible function and the collapsed function.

Summation in a finite Abelian group is a function which is known to be n-private [8]. In a summation, the effect of one party's input can be thought of as applying a permutation to the sum of the other parties' inputs. We generalize this by defining a permuted sum where we give one of the parties a special role and let her input select an arbitrary permutation to be applied to the sum of the other parties' inputs. All functions which are sums, i.e. can be rewritten as $\sum_{i=1}^{n} f_i(x_i)$, are also permuted sums. In our applications, the sum may be a Boolean sum or over $\mathbb{Z}_3$. We show two example functions which are permuted sums in Figure 3

Definition 6 (Permuted sum). *We say that a function is a* permuted sum *if it can be written as $\pi_{x_i}(\sum_{j \neq i} f_j(x_j))$ where π_x is a permutation. We refer to party i as the* permuter.

$$\begin{array}{cccccc} 0&0&1&1&2&2 \\ 1&2&0&2&0&1 \end{array}$$

(a) f

$$\begin{array}{ccc|ccc|ccc} 0&1&2&0&2&1&1&0&2 \\ 1&2&0&2&1&0&0&2&1 \\ 2&0&1&1&0&2&2&1&0 \end{array}$$

(b) g

Fig. 3. Two example permuted sums. In f, party 2 (selecting column) is the permuter selecting one of the 6 permutations. The function $g = \pi_{x_3}(x_1 + x_2)$ where π_1 is the identity permutation, $\pi_2 = (12)$ and $\pi_3 = (01)$.

With these definitions, we are now ready to state and prove our structure lemma:

Lemma 8 (Structure lemma). *For every normalized n-argument function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$, at least one of the following holds:*

- *f has an embedded* OR
- *f is a permuted sum*
- *f is collapsible*

We present protocols for n-privately evaluating permuted sums (Protocol 2) and collapsible functions (Protocol 3) which do not contain an embedded OR. In Theorem 22 we show that if f contains an embedded OR, it also contains a small embedded OR. This, together with Corollary 4 concludes the proof of our Theorem 1.

To prove the structure lemma, we perform a case-analysis based on a property of f we call a link:

Definition 7 (Link, link-free). *We say that an n-argument function has a* link *(*over output α*) in dimension k if there exists inputs $x, y, x \neq y$, and $\boldsymbol{a}$ such that $\alpha = f^{\boldsymbol{a}}_{\{k\}}(x) = f^{\boldsymbol{a}}_{\{k\}}(y)$. We say that f has* links in c dimensions *if there are precisely c distinct k such that f has a link in dimension k. We say that a function is* link-free *if it has no links.*

Lemma 9. *In a corner-free n-argument function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$, if there are links between inputs x and y in dimension k over two distinct outputs, then x and y are redundant for player k.*

Proof. Let f have links over α and β between inputs x and y in dimension k. That is, there exists values $\boldsymbol{a}$, $\boldsymbol{b}$ such that $f^{\boldsymbol{a}}_{\{k\}}(x) = f^{\boldsymbol{a}}_{\{k\}}(y) = \alpha$, and $f^{\boldsymbol{b}}_{\{k\}}(x) = f^{\boldsymbol{b}}_{\{k\}}(y) = \beta$. Suppose that for some $\boldsymbol{c}$ we have $f^{\boldsymbol{c}}_{\{k\}}(x) \neq f^{\boldsymbol{c}}_{\{k\}}(y)$. Then one of $f^{\boldsymbol{c}}_{\{k\}}(x)$ and $f^{\boldsymbol{c}}_{\{k\}}(y)$ equals α or β. If one of them is α then f has an embedded OR with $S_1 = \{k\}$, $S_2 = \{k\}^C$ using inputs (x, y) and $(\boldsymbol{a}, \boldsymbol{c})$. If one is β then f has an embedded OR with $S_1 = \{k\}$, $S_2 = \{k\}^C$ using inputs (x, y) and $(\boldsymbol{b}, \boldsymbol{c})$. $\square$

Looking at the proof of Lemma 9 we begin to see the importance of the small range of the function to the analysis. It also highlights the added complexities compared to the Boolean case, as for a Boolean function any link implies that two inputs are redundant. From the lemma and its proof follow two corollaries about normalized functions with range $\mathbb{Z}_3$:

Corollary 10. *For a normalized n-argument function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$ with a link over α in dimension k for inputs x, y, for all $\boldsymbol{a}$, we have that $f^{\boldsymbol{a}}_{\{k\}}(x)$ uniquely determines $f^{\boldsymbol{a}}_{\{k\}}(y)$. More specifically, the possible combinations of values are (α, α); (β, γ); (γ, β).*

Proof. Follows from the proof of Lemma 9. □

Corollary 11. *A normalized n-argument function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$ cannot have links over α in dimension k for inputs x, y, and x, z.*

Proof. By Corollary 10 the value at x determines the value at both y and z and hence inputs y and z are redundant. □

Analogously to an embedded OR, we introduce notation for the various 2×2 substructures in a function. Apart from the embedded OR, two of them feature prominently in our proofs. Firstly, a 2×2 substructure with one output occurring on the diagonal, and the two other values occurring once each on the opposite diagonal is called Aff_3. Secondly, a 2×2 substructure where one output is on one diagonal, and another is on the other is referred to as an XOR. For the XOR, we also define the type of an XOR as the pair (without order) of outputs in the XOR. All the substructures which can occur (up to symmetries) are depicted in Figure 4. A 2×2 substructure where only one output occurs is called constant, and if we want to emphasize that it is the output α which occurs, we write (α)-constant.

$\alpha\ \alpha$	$\alpha\ \alpha$	$\alpha\ \alpha$	$\alpha\ \alpha$	$\alpha\ \beta$	$\alpha\ \beta$
$\alpha\ \alpha$	$\alpha\ \beta$	$\beta\ \beta$	$\beta\ \gamma$	$\beta\ \alpha$	$\gamma\ \alpha$
(a) Constant	(b) OR	(c) 2-link	(d) Link	(e) XOR	(f) Aff_3

Fig. 4. The six 2×2 substructures.

Definition 8 (Type of an XOR). *If an XOR consists of outputs α and β we say that it is an XOR of type (α, β), denoted (α, β)-XOR. The order of elements is not important, so for functions to $\mathbb{Z}_3$ there are three possible types of XOR: (α, β), (α, γ), (β, γ).*

Our name Aff_3 comes from the fact that it can be expressed as an affine function modulo 3, analogously to the fact that XOR can be expressed as a sum modulo 2. We do not need that it is affine, but we make use of the fact that a function where all subfunctions are of the form Aff_3 can be written as a sum on the form $\sum_{i=1}^{n} f_i(x_i)$ with summation in $\mathbb{Z}_3$.

Lemma 12. *An n-argument corner-free function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$ such that all 2×2 subfunctions are of the form Aff_3 can be expressed as $\sum_{i=1}^{n} f_i(x_i)$ with summation in $\mathbb{Z}_3$.*

Proof. By Lemma 7 we need to verify that (1) holds for all 2×2 subfunctions, which are all of the form Aff_3. For all ways of assigning 0, 1 and 2 (distinctly) to α, β, γ we have that $2\alpha \equiv \beta + \gamma \pmod 3$. As $2 \equiv -1 \pmod 3$ this is equivalent to $\alpha + \beta + \gamma \equiv 0 \pmod 3$. □

In our proof of Lemma 8 we consider the substructures occurring in f. We begin by establishing three preliminary lemmas. The lemmas come into play primarily in cases when f contains links in few dimensions (none or one), and if f has an XOR spanned by dimensions i, j and f is link-free in those dimensions, then $|A_i| = |A_j| = 2$, giving some intuition for the condition on the size of the two inputs in the lemmas. We highlight the proof idea for each of the lemmas and give full proofs in the appendix.

Lemma 13. *Let f be an n-argument corner-free function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$ with i, j such that $|A_i| = |A_j| = 2$ such that for all $\boldsymbol{a}$, $f^{\boldsymbol{a}}_{\{i,j\}}$ is an XOR. If all three types of XOR's occur then there is a dimension k such that the input in dimension k determines the type of XOR.*

Proof (Idea). We show that if no such k exists then f contains an embedded OR. Full proof in Section A.2. □

Lemma 14. *Let f be an n-argument corner-free function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$ with i, j such that $|A_i| = |A_j| = 2$ and an output α such that for all $\boldsymbol{a}$ precisely one diagonal of $f^{\boldsymbol{a}}_{\{i,j\}}$ has two α's. Then f is collapsible.*

Proof (Idea). If f is not collapsible then the collapsed function contains an embedded OR. We show that this implies an embedded OR in f as well. Full proof in Section A.3. □

Lemma 15. *An n-argument corner-free function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$ with i, j such that $|A_i| = |A_j| = 2$ and such that for some $\boldsymbol{a}$, $f^{\boldsymbol{a}}_{\{i,j\}}$ is an Aff_3 and for some $\boldsymbol{b}$, $f^{\boldsymbol{b}}_{\{i,j\}}$ is an XOR is collapsible.*

Proof (Idea). We prove that f fulfills the conditions of Lemma 14. Full proof in Section A.4. □

Our proof of Lemma 8 proceeds in three separate lemmas, depending on whether the function f is link-free (Lemma 16), has links in one dimension (Lemma 17), or if it has links in two or more dimensions (Lemma 18). As the proofs are long and consist mainly of case analysis, we give them in the appendix and simply state the lemmas here.

Lemma 16. *Every n-argument link-free, corner-free function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$ is collapsible or a permuted sum.*

Proof (Idea). Case analysis showing we can apply one of Lemma 13, Lemma 14 and Lemma 15. Full proof in Section A.5 □

Lemma 17. *Every n-argument function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$ with links in 1 dimension and without an embedded* OR *is collapsible or a permuted sum.*

Proof (Idea). Case analysis showing we can apply one of Lemma 13, Lemma 14 and Lemma 15. Full proof in Section A.6. □

Lemma 18. *Every n-argument function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$ with links in 2 or more dimensions and without an embedded* OR *is collapsible.*

Proof (Idea). We show that all links must be over the same output. This gives some implications for the substructures of f which we use to show f must be collapsible. Full proof in Section A.7. □

4 Protocols

With the structure lemma established, we can now turn to the question of n-private protocols for collapsible functions and permuted sums. From the definitions of the two classes, we have two natural and easy protocols. The main problem we need to address in this section is proving the existence of a protocol for collapsible functions. For a function which is collapsible by collapsing β and γ it is clear from the definition that we can n-privately evaluate if the output is α or if it is one of β and γ. The key issue is to prove that we can then proceed with a second step where we can n-privately evaluate whether the output is β or if it is γ.

The construction of this second step relies on the passive model of adversaries and the knowledge that the output of the function is not α. Thus, in our second step we compute a sum which may have different outputs at points where the original function had α's. Such a construction is inherently insecure with active adversaries, as they may switch inputs between the first step of the decomposition and the second and would then learn some information about the other parties' inputs.

In both of our protocols we use a subprotocol by Chor and Kushilevitz [8] for n-private summation over any finite Abelian group. For completeness, we include a description of their protocol as Protocol 1. When used in our protocol for a permuted sum, the summation is either Boolean or in $\mathbb{Z}_3$ depending on the function f (but not on the inputs).

Protocol 1 (Summation [8]). The protocol for summation where party P_i participates with input x_i proceeds as follows:

1. In round $1 \leq i \leq n-2$, party P_i sums all its received messages, $w_i = \sum_{j=1}^{i-1} z_{j,i}$. Then, it chooses random group elements $z_{i,i+1}, z_{i,i+2}, \ldots, z_{i,n-1}$. Finally, it computes $z_{i,n}$ such that $x_i + w_i = \sum_{j=i+1}^{n} z_{i,j}$ and sends $z_{i,j}$ to P_j $(j > i)$.

2. In round $n-1$, party P_{n-1} computes $z_{n-1,n} = x_{n-1} + \sum_{j=1}^{n-2} z_{j,n-1}$ and sends $z_{n-1,n}$ to P_n.
3. In round n, party P_n computes the sum s as $s = x_n + \sum_{j=1}^{n-1} z_{j,n}$.

All sums are computed over some fixed finite Abelian group.

Protocol 2 (Permuted sum). The protocol for evaluating a permuted sum f, where party P_i (without loss of generality we assume the permuter is party n) participates with input x_i proceeds as follows:

1. Use Protocol 1 to privately compute $s = \sum_{j=1}^{n-1} f_j(x_j)$ such that only the permuter learns s.
2. The permuter computes the output as $\pi_{x_n}(s)$ and sends it to the other parties.

The sum is computed modulo 2, or 3, depending on f.

Protocol 3 (Collapsible). The protocol for evaluating a function f collapsible with partition $R' = \{\gamma\}$, where party P_i participates with input x_i proceeds as follows:

1. Use Protocol 1 to compute $s = \sum_{i=1}^{n} f_i(x_i) \pmod 2$, with f_i such that $s = 1$ iff $f(\boldsymbol{x}) = \gamma$
2. If $s = 0$, compute $s' = \sum_{i=1}^{n} g_i(x_i) \pmod 4$, with g_i such that $f(\boldsymbol{x}) = \alpha$ implies $s' = 0$, and $f(\boldsymbol{x}) = \beta$ implies $s' = 2$.

The correctness of Protocol 2 follows immediately from the definition of a permuted sum. In Protocol 3, since f is collapsible, the functions f_i exist by the definition of a collapsible function. However, the existence of appropriate g_i is not as straightforward. We prove, constructively, in Lemma 21 that they always exist for corner-free collapsible functions with range $\mathbb{Z}_3$. We stress that the choice of g_i does not depend on the input $\boldsymbol{x}$, but only on the function f.

The privacy of both these protocols is straightforward, and we only sketch the arguments.

Theorem 19. *Protocol 2 is n-private.*

Proof. The subprotocol used for summation was proven to be n-private in [8]. Due to the structure of the function, we see that the permuter, P_n, learns the sum s from $f(\boldsymbol{x})$ and x_n, since $s = \pi_{x_n}^{-1}(f(\boldsymbol{x}))$. □

Theorem 20. *Protocol 3 is n-private.*

Proof. The subprotocol used for summation was proven to be n-private in [8]. When the output is γ then, by the privacy of the summation sub-protocol, the protocol is private. Furthermore, when the output is one of α, β, then the privacy of the composed protocol also follow directly from the privacy of the subprotocols. The first sum only reveals that the output is one of α, β, and then, the condition on g_i is sufficient to guarantee that the sum s' reveals nothing but whether the output is α or β, as with a passive adversary we are guaranteed that s' is either 0 or 2. □

While the privacy is straightforward, the proof that there are functions g_i as required by Protocol 3 is rather involved and we simply state the lemma here and give the proof in the appendix. One may intuitively expect that such functions could simply be Boolean, but it turns out that for some f we do need the full range of $\mathbb{Z}_4$.

Lemma 21. *Protocol 3 can evaluate all corner-free, collapsible functions with range* $\mathbb{Z}_3$.

Proof (Idea). We construct a function g such that $f(\boldsymbol{x}) = \alpha \implies g(\boldsymbol{x}) = 0$ and $f(\boldsymbol{x}) = \beta \implies g(\boldsymbol{x}) = 2$. By case analysis on the induced rectangles in g, we show that g satisfies the conditions of Lemma 7 and hence there are g_i as required by Protocol 3. Full proof in Section A.8. $\square$

5 An Embedded OR Implies a Small Embedded OR

Previously, we have often assumed that functions are free of embedded OR's of *any* degree (i.e., that they are corner-free). However, to be able to apply Corollary 4 we need to show that a sufficiently small embedded OR exists.

For Boolean functions f, if f has an embedded OR of any degree, then it also has an embedded OR of degree 1, as proved in [11], explaining the zero-one nature of Boolean privacy.

It turns out that for functions with range $\mathbb{Z}_3$, similarly to the Boolean case, the presence of a large embedded OR implies that the function also contains a small one. We state the theorem here and give the proof in the appendix.

Theorem 22. *Every n-argument function $f : A_1 \times \ldots A_n \to \mathbb{Z}_3$ that has an embedded* OR *of any degree has an embedded* OR *of degree at most* 3. *Furthermore, every* 4*-argument function $f : A_1 \times A_2 \times A_3 \times A_4 \to \mathbb{Z}_3$ that has an embedded* OR*, also has one of degree at most* 2.

Proof (Idea). The basic idea is similar to that used in the proof of Theorem 6. However, while the boolean case is fairly straightforward, our proof results in a fairly extensive case analysis. Full proof in Section A.9. $\square$

6 Proof of the Main Theorem

We now conclude by re-stating our main theorem and presenting the proof.

Theorem 1 (Main theorem). *For every n-argument function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$, f is either n-private, or it is $\lfloor (n-1)/2 \rfloor$-private and not $\lceil n/2 \rceil$-private.*

Proof. If f is corner-free, then by Lemma 8 it is a permuted sum, collapsible, or both. Thus, it can be n-privately computed by Protocol 2 or Protocol 3.

If f is not corner-free, then by Theorem 22 it contains an embedded OR of degree at most $\lceil n/2 \rceil$. Thus, by Corollary 4, f is not $\lceil n/2 \rceil$-private. $\square$

Acknowledgements

I would like to thank Johan Håstad for his many helpful insights, Dominik Raub for introducing me to this problem and for the interesting discussions, and Per Austrin for Protocol 3.

References

1. Donald Beaver. Perfect privacy for two-party protocols. In J. Feigenbaum and M. Merritt, editors, *Proceedings of DIMACS Workshop on Distributed Computing and Cryptology*, volume 2, pages 65–77. American Mathematical Society, 1989.
2. Zuzana Beerliová-Trubíniová, Matthias Fitzi, Martin Hirt, Ueli M. Maurer, and Vassilis Zikas. MPC vs. SFE: Perfect security in a unified corruption model. In Ran Canetti, editor, *TCC*, volume 4948 of *Lecture Notes in Computer Science*, pages 231–250. Springer, 2008.
3. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10. ACM, 1988.
4. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *STOC*, pages 11–19. ACM, 1988.
5. Benny Chor, Mihály Geréb-Graus, and Eyal Kushilevitz. On the structure of the privacy hierarchy. *J. Cryptology*, 7(1):53–60, 1994.
6. Benny Chor and Yuval Ishai. On privacy and partition arguments. *Inf. Comput.*, 167(1):2–9, 2001.
7. Benny Chor and Eyal Kushilevitz. A zero-one law for boolean privacy. *SIAM J. Discrete Math.*, 4(1):36–47, 1991.
8. Benny Chor and Eyal Kushilevitz. A communication-privacy tradeoff for modular addition. *Inf. Process. Lett.*, 45(4):205–210, 1993.
9. Benny Chor and Netta Shani. The privacy of dense symmetric functions. *Computational Complexity*, 5(1):43–59, 1995.
10. Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *J. Cryptology*, 13(1):31–60, 2000.
11. Joe Kilian, Eyal Kushilevitz, Silvio Micali, and Rafail Ostrovsky. Reducibility and completeness in private computations. *SIAM J. Comput.*, 29(4):1189–1208, 2000.
12. Robin Künzler, Jörn Müller-Quade, and Dominik Raub. Secure computability of functions in the IT setting with dishonest majority and applications to long-term security. In Omer Reingold, editor, *TCC*, volume 5444 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009.
13. Eyal Kushilevitz. Privacy and communication complexity. *SIAM J. Discrete Math.*, 5(2):273–284, 1992.
14. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE, 1982.

A Proofs

A.1 Proof of Lemma 7

Lemma 7. *If an n-argument function $f : A_1 \times \ldots \times A_n \to G$, where G is an Abelian group, has the property that for every pair of dimensions j, k and inputs*

$x_1, x_2, y_1, y_2, \boldsymbol{a}$ *the following equivalence holds:*

$$f^{\boldsymbol{a}}_{\{j,k\}}(x_1, y_1) + f^{\boldsymbol{a}}_{\{j,k\}}(x_2, y_2) = f^{\boldsymbol{a}}_{\{j,k\}}(x_1, y_2) + f^{\boldsymbol{a}}_{\{j,k\}}(x_2, y_1), \tag{1}$$

then f can be rewritten as $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} f_i(x_i)$.

Proof. Induction on n. Rewrite (1) as

$$f^{\boldsymbol{a}}_{\{j,k\}}(x_1, y_1) - f^{\boldsymbol{a}}_{\{j,k\}}(x_1, y_2) = f^{\boldsymbol{a}}_{\{j,k\}}(x_2, y_1) - f^{\boldsymbol{a}}_{\{j,k\}}(x_2, y_2)$$

with $k = n$. This says that the function

$$g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, y_1) - f(x_1, \ldots, x_{n-1}, y_2)$$

does not change value when you change one arbitrary input in an arbitrary way. This implies that g is a constant function. The lemma now follows by induction on n.

□

A.2 Proof of Lemma 13

Lemma 13. *Let f be an n-argument corner-free function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$ with i, j such that $|A_i| = |A_j| = 2$ such that for all $\boldsymbol{a}$, $f^{\boldsymbol{a}}_{\{i,j\}}$ is an* XOR. *If all three types of* XOR*'s occur then there is a dimension k such that the input in dimension k determines the type of* XOR.

Proof. We assume that there is an (α, β)-XOR and an (α, γ)-XOR at Hamming distance 1. We denote the dimension by which they differ by k and relabel the inputs in dimension k such that $f^{\boldsymbol{a}}_{\{i,j,k\}}(\cdot, \cdot, 1)$ is an (α, β)-XOR and $f^{\boldsymbol{a}}_{\{i,j,k\}}(\cdot, \cdot, 2)$ is an (α, γ)-XOR.

We proceed to show that there is no $\boldsymbol{b}$ such that $f^{\boldsymbol{b}}_{\{i,j,k\}}(\cdot, \cdot, 1)$ or $f^{\boldsymbol{b}}_{\{i,j,k\}}(\cdot, \cdot, 2)$ is a (β, γ)-XOR. Assume to the contrary that there is a $\boldsymbol{b}$ such that $f^{\boldsymbol{b}}_{\{i,j,k\}}(\cdot, \cdot, 1)$ is a (β, γ)-OR (the case if it occurs at input 2 in dimension k is analogous). We illustrate this case in Figure 5 where we for simplicity show $\boldsymbol{b}$ as differing from $\boldsymbol{a}$ in only one dimension, which is not something we assume in the proof.

$$\begin{array}{cc|cc} \alpha & \beta & \alpha & \gamma \\ \beta & \alpha & \gamma & \alpha \\ \hline \gamma & \beta & & \\ \beta & \gamma & & \end{array}$$

Fig. 5. Illustration of a contradiction in the proof of Lemma 13.

What values can the function take at $f^{\boldsymbol{b}}_{\{i,j,k\}}(1, 1, 2)$? We claim that any output at that position would violate the assumption that f is corner-free. In

Figure 5 we can see why this is true for a simple function (writing α, β or γ anywhere in the missing 2×2 field can be verified to result in an embedded OR). We proceed with three almost identical cases (differing only in the interdependency of the coordinates, the core idea is captured by Figure 5):

Case 1: $f^{\boldsymbol{b}}_{\{i,j,k\}}(1,1,2) = \alpha$. We can find y (equal to 1 or 2) such that $f^{\boldsymbol{a}}_{\{i,j,k\}}(1,y,2) = \alpha$ as $f^{\boldsymbol{a}}_{\{i,j,k\}}(\cdot,\cdot,2)$ is an (α,γ)-XOR. We can also find x such that $f^{\boldsymbol{a}}_{\{i,j,k\}}(x,y,1) = \alpha$ as $f^{\boldsymbol{a}}_{\{i,j,k\}}(\cdot,\cdot,1)$ is an (α,β)-XOR. However, as $f^{\boldsymbol{b}}_{\{i,j,k\}}(\cdot,\cdot,1)$ is a (β,γ)-XOR we are guaranteed that $f^{\boldsymbol{b}}_{\{i,j,k\}}(x,1,1) \neq \alpha$. Thus, f contains an embedded OR with $S_1 = \{i,k\}$ and $S_2 = S_1^C$ using $(1,2);(x,1)$ on S_1 and $y;1$ or j and $\boldsymbol{a};\boldsymbol{b}$ on the rest of S_2.

Case 2: $f^{\boldsymbol{b}}_{\{i,j,k\}}(1,1,2) = \beta$. We can find x (equal to 1 or 2) such that $f^{\boldsymbol{b}}_{\{i,j,k\}}(x,1,1) = \beta$ as $f^{\boldsymbol{b}}_{\{i,j,k\}}(\cdot,\cdot,1)$ is an (β,γ)-XOR. We can also find y such that $f^{\boldsymbol{a}}_{\{i,j,k\}}(x,y,1) = \beta$ as $f^{\boldsymbol{a}}_{\{i,j,k\}}(\cdot,\cdot,1)$ is an (α,β)-XOR. However, as $f^{\boldsymbol{a}}_{\{i,j,k\}}(\cdot,\cdot,2)$ is a (α,γ)-XOR we are guaranteed that $f^{\boldsymbol{a}}_{\{i,j,k\}}(1,y,2) \neq \beta$. Thus, f contains an embedded OR with $S_1 = \{i,k\}$ and $S_2 = S_1^C$ using $(1,2);(x,1)$ on S_1 and $y;1$ or j and $\boldsymbol{a};\boldsymbol{b}$ on the rest of S_2.

Case 3: $f^{\boldsymbol{b}}_{\{i,j,k\}}(1,1,2) = \gamma$. We can find x (equal to 1 or 2) such that $f^{\boldsymbol{b}}_{\{i,j,k\}}(x,1,1) = \gamma$ as $f^{\boldsymbol{b}}_{\{i,j,k\}}(\cdot,\cdot,1)$ is an (β,γ)-XOR. We can also find y such that $f^{\boldsymbol{a}}_{\{i,j,k\}}(1,y,2) = \gamma$ as $f^{\boldsymbol{a}}_{\{i,j,k\}}(\cdot,\cdot,2)$ is an (α,γ)-XOR. However, as $f^{\boldsymbol{a}}_{\{i,j,k\}}(\cdot,\cdot,1)$ is a (α,β)-XOR we are guaranteed that $f^{\boldsymbol{a}}_{\{i,j,k\}}(x,y,1) \neq \gamma$. Thus, f contains an embedded OR with $S_1 = \{i,k\}$ and $S_2 = S_1^C$ using $(1,2);(x,1)$ on S_1 and $y;1$ or j and $\boldsymbol{a};\boldsymbol{b}$ on the rest of S_2.

We now conclude that there is no $\boldsymbol{b}$ such that $f^{\boldsymbol{b}}_{\{i,j,k\}}(\cdot,\cdot,1)$ or $f^{\boldsymbol{b}}_{\{i,j,k\}}(\cdot,\cdot,2)$ is a (β,γ)-XOR. As f has all types of XOR's, there must still be a (β,γ)-XOR in the function. Thus, we see that $|A_k| \geq 3$, and we can assume there is a $\boldsymbol{b}$ such that $f^{\boldsymbol{b}}_{\{i,j,k\}}(\cdot,\cdot,3)$ is a (β,γ)-XOR.

We claim that $f^{\boldsymbol{a}}_{\{i,j,k\}}(\cdot,\cdot,3)$ must be a (β,γ)-XOR. To see this we observe that if it was another type of XOR, then by the same proof that showed that there is no (β,γ)-XOR for $k = 1,2$ we could have shown that there was not (β,γ)-XOR for $k = 3$, but we know that $f^{\boldsymbol{b}}_{\{i,j,k\}}(\cdot,\cdot,3)$ is a (β,γ)-XOR. We now see that for a given x_k, all XOR's must be of the same type as that at $f^{\boldsymbol{a}}_{\{i,j,x_k\}}(\cdot,\cdot,k)$ which concludes our proof. □

A.3 Proof of Lemma 14

Lemma 14. *Let f be an n-argument corner-free function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$ with i,j such that $|A_i| = |A_j| = 2$ and an output α such that for all $\boldsymbol{a}$ precisely one diagonal of $f^{\boldsymbol{a}}_{\{i,j\}}$ has two α's. Then f is collapsible.*

Proof. We claim that f is collapsible by collapsing β and γ. To prove this we show that the collapsed function

$$g(\boldsymbol{x}) = \begin{cases} 1 & \text{if } f(\boldsymbol{x}) \in \{\beta,\gamma\} \\ 0 & \text{if } f(\boldsymbol{x}) = \alpha \end{cases}$$

does not contain an embedded OR of degree 1. Then by Theorem 6 we have that g is corner-free and Theorem 5 implies that the collapsed function can be written as a Boolean sum.

We begin by observing that as each 2×2 plane spanned by dimensions i, j contains exactly one diagonal with α's, each such 2×2 plane contains two α's, as if it had three α's it would be an embedded OR and if it had four α's both diagonals would have two α's. We further make the observation that a pair of neighboring outputs in dimension i (and analogously in j) are such that exactly one of them is α. More formally, for all $\boldsymbol{c}$ if $f^{\boldsymbol{c}}_{\{i\}}(1) = \alpha$ then $f^{\boldsymbol{c}}_{\{i\}}(2) \neq \alpha$ and if $f^{\boldsymbol{c}}_{\{i\}}(1) \neq \alpha$ then $f^{\boldsymbol{c}}_{\{i\}}(2) = \alpha$.

As g is Boolean, by Theorem 6 we know that if g has an embedded OR (of any degree), it also has an embedded OR of degree 1. We assume by contradiction that there is an embedded OR of degree 1 in g. We reorder inputs and dimensions such that the embedded OR is spanned by dimensions $1, 2$ using inputs $(1, 2); (1, 2)$, with other inputs as $\boldsymbol{a}$. We say that $g^{\boldsymbol{a}}_{\{1,2\}}$ is an embedded OR with slight abuse of notation (as $|A_1|$ or $|A_2|$ could be greater than 2). We see that the embedded OR cannot have three 1's as g takes the value 0 where f takes the value α, so an embedded OR with three 0's corresponds to an embedded OR with three α in f, which is corner-free. Thus, the embedded OR must have three 1's.

From our observation we know that each 2×2 plane in g spanned by i, j has two 0's, so there cannot be an OR in g with three 1's spanned by dimensions i, j. Thus, at least one of i and j must be different from both 1 and 2. We assume $i \neq 1, 2$ and reorder inputs such that the embedded OR occurs when $x_i = 1$. Let $\boldsymbol{b}$ be $\boldsymbol{a}$ with the value at x_i removed.

We now consider what values occur at $f^{\boldsymbol{b}}_{\{1,2,i\}}(\cdot, \cdot, 2)$. We know that of the four outputs of $f^{\boldsymbol{b}}_{\{1,2,i\}}(\cdot, \cdot, 1)$ one is α and three are different from α. But by our observation, this implies that of the four outputs of $f^{\boldsymbol{b}}_{\{1,2,i\}}(\cdot, \cdot, 2)$ three are α and one is different from α. This concludes our proof as it shows that an embedded OR in g implies an embedded OR in f which we assumed to be corner-free. $\square$

A.4 Proof of Lemma 15

Lemma 15. *An n-argument corner-free function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$ with i, j such that $|A_i| = |A_j| = 2$ and such that for some $\boldsymbol{a}$, $f^{\boldsymbol{a}}_{\{i,j\}}$ is an Aff_3 and for some $\boldsymbol{b}$, $f^{\boldsymbol{b}}_{\{i,j\}}$ is an XOR is collapsible.*

Proof. Let the output that appears twice in $f^{\boldsymbol{a}}_{\{i,j\}}$ be α, and reorder inputs such that $f^{\boldsymbol{a}}_{\{i,j\}}(1, 1) = f^{\boldsymbol{a}}_{\{i,j\}}(2, 2) = \alpha$.

We now claim that $f^{\boldsymbol{b}}_{\{i,j\}}(1, 2) = f^{\boldsymbol{b}}_{\{i,j\}}(2, 1) = \alpha$. As $f^{\boldsymbol{b}}_{\{i,j\}}$ is an XOR we know that $f^{\boldsymbol{b}}_{\{i,j\}}(1, 2) = f^{\boldsymbol{b}}_{\{i,j\}}(2, 1)$. If $f^{\boldsymbol{b}}_{\{i,j\}}(1, 2) = f^{\boldsymbol{b}}_{\{i,j\}}(2, 1) \in \{\beta, \gamma\}$ then there is an embedded OR with $S_1 = \{i, j\}$ and $S_2 = S_1^C$ as using inputs $(1, 2); (2, 1)$ on S_1 and $\boldsymbol{a}; \boldsymbol{b}$ on S_2. We assume the other diagonal of the XOR consists of β's, i.e. $f^{\boldsymbol{b}}_{\{i,j\}}(1, 1) = f^{\boldsymbol{b}}_{\{i,j\}}(2, 2) = \beta$, and that $f^{\boldsymbol{a}}_{\{i,j\}}(1, 2) = \beta$, $f^{\boldsymbol{a}}_{\{i,j\}}(2, 1) = \gamma$. This is without loss of generality as we can relabel outputs and switch the roles of parties 1 and 2.

$$\begin{array}{cc} \boldsymbol{\alpha} & \beta \\ \gamma & \alpha \end{array} \qquad \begin{array}{cc} \beta & \boldsymbol{\alpha} \\ \alpha & \boldsymbol{\beta} \end{array}$$

(a) $f^{\boldsymbol{a}}_{\{i,j\}}$ (b) $f^{\boldsymbol{b}}_{\{i,j\}}$

Fig. 6. An XOR and Aff_3 in f. The outputs involved in proving that f has no links in dimension i are highlighted.

We claim that the function f cannot have any links in dimensions i or j. To see this for dimension i, we see that $f^{\boldsymbol{a}}_{\{i,j\}}(1,1) = \alpha$ and $f^{\boldsymbol{a}}_{\{i,j\}}(2,1) = \gamma$ but also $f^{\boldsymbol{b}}_{\{i,j\}}(1,2) = \alpha$ and $f^{\boldsymbol{b}}_{\{i,j\}}(2,2) = \beta$. Thus, the value of $f^{\boldsymbol{c}}_{\{i\}}(2)$ is not a function of the value of $f^{\boldsymbol{c}}_{\{i\}}(1)$ for all $\boldsymbol{c}$ and the contrapositive form of Corollary 10 gives that f cannot have a link between inputs 1 and 2 in dimension i. Similarly for dimension j, we have that $f^{\boldsymbol{a}}_{\{i,j\}}(2,2) = \alpha$ and $f^{\boldsymbol{a}}_{\{i,j\}}(2,1) = \gamma$, but also $f^{\boldsymbol{b}}_{\{i,j\}}(1,2) = \alpha$ and $f^{\boldsymbol{b}}_{\{i,j\}}(1,1) = \beta$. This demonstrates that $f^{\boldsymbol{c}}_{\{j\}}(1)$ is not a function of $f^{\boldsymbol{c}}_{\{j\}}(2)$ for all $\boldsymbol{c}$, and by the contrapositive form of Corollary 10, there is no link between inputs 2 and 1 in dimension j.

We proceed by proving that for all $\boldsymbol{c}$ precisely one of the two diagonals of $f^{\boldsymbol{c}}_{\{i,j\}}$ contains two α's. What are the possible values for $(f^{\boldsymbol{c}}_{\{i,j\}}(1,2), f^{\boldsymbol{c}}_{\{i,j\}}(2,1))$? We proved (when $\boldsymbol{c} = \boldsymbol{b}$ but we made no use of any properties of $\boldsymbol{b}$) that they cannot be (β,β) or (γ,γ). Furthermore, as $f^{\boldsymbol{b}}_{\{i,j\}}(1,2) = f^{\boldsymbol{b}}_{\{i,j\}}(2,1) = \alpha$ it cannot be that precisely one of the values is α, as then f would have an embedded OR with $S_1 = \{i,j\}$ and $S_2 = S_1^C$ using inputs $(1,2);(2,1)$ on S_1 and $\boldsymbol{b};\boldsymbol{c}$ on S_2. Thus the only remaining possibilities are $(\alpha,\alpha); (\beta,\gamma); (\gamma,\beta)$. As f has no links in dimension i or j we see that in the first case neither $f^{\boldsymbol{c}}_{\{i,j\}}(1,1)$ nor $f^{\boldsymbol{c}}_{\{i,j\}}(2,2)$ can equal α. In the two latter cases we have again by the link-freeness in dimensions i and j that $f^{\boldsymbol{c}}_{\{i,j\}}(1,1) = f^{\boldsymbol{c}}_{\{i,j\}}(2,2) = \alpha$. By Lemma 14 we have that f is collapsible as claimed. □

A.5 Proof of Lemma 16

Lemma 16. *Every n-argument link-free, corner-free function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$ is collapsible or a permuted sum.*

Proof. For a link-free and corner-free function, only two possibilities remain for the structure of an induced rectangle: it can either be an XOR or an Aff_3. If f contains an XOR spanned by dimensions i and j, then $|A_i| = |A_j| = 2$ since f is link-free.

We proceed with a case analysis. If f does not contain an XOR, then we select the first case. Otherwise, we pick an arbitrary XOR occurring in f and fix the dimensions i and j spanning it, and denote by $\boldsymbol{a}$ a set of inputs such that $f^{\boldsymbol{a}}_{\{i,j\}}$ is an (α,β)-XOR (if f has an XOR, we can relabel outputs such that there is an (α,β)-XOR). When we have fixed dimensions i,j we select one of the four last cases of our proof based *only* on the 2×2-planes spanned by dimensions i and j.

Case 1: Only Aff_3 (all dimensions). If all induced rectangles of f are of the form Aff_3, then f satisfies the condition of Lemma 12 and is a sum, and thus also a permuted sum.

Case 2: Both XOR and Aff_3 (spanned by i, j). By Lemma 15 we have that$f$ is collapsible.

Case 3: Only XOR, one type of XOR (spanned by i, j). If only one type of XOR's occur, then f is a Boolean corner-free function and by Theorem 5 we have that f is a sum, and thus also a permuted sum.

Case 4: Only XOR, two types of XOR (spanned by i, j). We assume that f contains XOR's of types (α, β) and (α, γ). Then α occurs on exactly one diagonal of all 2×2 planes spanned by dimensions i, j and by Lemma 14 f is collapsible by collapsing β and γ.

Case 5: Only XOR, three types of XOR (spanned by i, j). By Lemma 13 we see that there must be a dimension k such that the input in dimension k determines the type of the XOR. Reorder inputs such that for input 1 in dimension k the 2×2-planes spanned by i and j are (α, β)-XOR's. We let $\boldsymbol{a} = (1)$ and $S_1 = \{k\}^C$ and see that $f^{\boldsymbol{a}}_{\{S_1\}}$ is a Boolean corner-free function. Thus, Theorem 5 implies that $f^{\boldsymbol{a}}_{\{S_1\}}(x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_n) = \sum_{i \neq k} f_i(x_i)$ with the sum computed modulo 2.

We claim that f is a permuted sum with P_k as the permuter and the sum computed modulo 2. To see this, we prove that for all $x_k \in A_k$ and for all inputs $\boldsymbol{b}$ we have $f^{\boldsymbol{b}}_{\{k\}}(x_k) = \pi_{x_k}\{f^{\boldsymbol{b}}_{\{k\}}(1)\}$. As f is link-free, we have that $f^{\boldsymbol{b}}_{\{k\}}(x_k) \neq f^{\boldsymbol{b}}_{\{k\}}(1)$. If x_k is such that the 2×2-planes spanned by dimensions 1 and 2 when the input in dimension k is x_k are (α, β)-XOR then this means that $f^{\boldsymbol{b}}_{\{k\}}(1) = \alpha \implies f^{\boldsymbol{b}}_{\{k\}}(x_k) = \beta$ and $f^{\boldsymbol{b}}_{\{k\}}(1) = \beta \implies f^{\boldsymbol{b}}_{\{k\}}(x_k) = \alpha$. Similarly if the XOR's are (α, γ)-XOR's $f^{\boldsymbol{b}}_{\{k\}}(1) = \alpha \implies f^{\boldsymbol{b}}_{\{k\}}(x_k) = \gamma$, and as the 2×2-planes are XOR's we have $f^{\boldsymbol{b}}_{\{k\}}(1) \neq \alpha \implies f^{\boldsymbol{b}}_{\{k\}}(x_k) = \alpha$. The case for x_k with (β, γ)-XOR's is analogous, concluding the proof. □

A.6 Proof of Lemma 17

Lemma 17. *Every n-argument function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$ with links in 1 dimension and without an embedded* OR *is collapsible or a permuted sum.*

Proof. For convenience of notation, we reorder parties and inputs such that there is a link between inputs 1 and 2 in dimension 1 over output α. We consider the functions $g_m(x_2 \ldots, x_n) = f(m, x_2, \ldots, x_n)$. As f has links only in 1 dimension, each g_m is link-free. From Corollary 10, we have that $g_2 = \pi_2 \circ g_1$ where π_2 is a permutation (transposing β and γ). If there is a link between inputs m and m' in dimension 1 we say there is a link between g_m and $g_{m'}$, with slight abuse of notation.

As g_1 is corner-free and link-free, there are only two possible 2×2 structures which can occur: Aff_3 and XOR. If an XOR occurs as a substructure spanned by dimensions i, j, then since g_1 is link-free we must have $|A_i| = |A_j| = 2$. Our proof proceeds in five cases depending on the structures in g_1 (but not on f as a whole). As in the proof of Lemma 16, in the four cases when g_1 has an XOR we fix dimensions i, j spanning an XOR and then select case based *only* on the 2×2 substructures spanned by dimensions i and j. Our cases are:

1. g_1 without XOR (all dimensions)
2. g_1 with both XOR and Aff_3 (spanned by i, j)
3. g_1 with only XOR's, one type of XOR (spanned by i, j)
4. g_1 with only XOR's, two types of XOR (spanned by i, j)
5. g_1 with only XOR's, three types of XOR (spanned by i, j)

Case 1: g_1 without XOR (all dimensions). We begin with the case that g_1 does not contain an XOR, and thus only consists of Aff_3. As $g_2 = \pi_2 \circ g_1$ the same is true for g_2. As all substructures of g_1 are of the form Aff_3, we can apply Lemma 12 to see that $g_1 = \sum_{k=2}^{n} f_k(x_k)$ where the sum is over $\mathbb{Z}_3$. Thus, if $g_m = \pi_m \circ g_1$ for all m, then f is a permuted sum with P_1 as the permuter. By Corollary 10 we know that if there is a link between g_1 and g_m or g_2 and g_m then $g_m = \pi_m \circ g_1$ as desired. This will be the case for all but a special case (when f is collapsible). The remainder of this proof assumes f is not a permuted sum and, after proving many restrictions on such f, shows that it is then collapsible by collapsing β and γ.

If f is not a permuted sum then there is an m such that g_m cannot be written on the form $g_m = \pi_m \circ g_1$. We claim that in this case, all Aff_3 in g_1 must have two α (the output linking g_1 and g_2). To see this, consider an Aff_3 in g_1 with only one α. Let it be spanned by some dimensions i, j and occur at inputs $\boldsymbol{a}$. We say that ${g_1}^{\boldsymbol{a}}_{\{i,j\}}$ is an Aff_3 (with slight abuse of notation as $|A_i|$ or $|A_j|$ may be greater than 2). As π_2 transposes β and γ, ${g_2}^{\boldsymbol{a}}_{\{i,j\}}$ is also an Aff_3 with one α. But this implies that for g_m not to have a link to either g_1 or g_2 then ${g_m}^{\boldsymbol{a}}_{\{i,j\}}$ would have to take the value α at precisely three points, giving an embedded OR. We illustrate this case in Figure 7. We claim that this also means that in this case there can be no $m' > 2$ such that $g_{m'}$ has a link to g_1 or g_2. By Corollary 11 the link would have to be over an output different from α which results in a situation analogous to an Aff_3 with only one α.

$$\begin{array}{cc|cc|cc} \beta & \alpha & \gamma & \alpha & \cdot & \cdot \\ \gamma & \beta & \beta & \gamma & \cdot & \cdot \end{array}$$

Fig. 7. A contradiction in the proof of Lemma 17, showing (a part of) g_1 to the left, g_2 in the middle, and g_m to the right.

Fix two distinct dimensions i, j different from 1. We now show that $|A_i| = |A_j| = 2$. For some inputs $\boldsymbol{a}$ we know that ${g_1}^{\boldsymbol{a}}_{\{i,j\}}$ is an Aff_3 with two α (again, with slight abuse of notation as we have not yet shown $|A_i| = |A_j| = 2$). Reorder inputs such that ${g_1}^{\boldsymbol{a}}_{\{i,j\}}(1,1) = {g_1}^{\boldsymbol{a}}_{\{i,j\}}(2,2) = \alpha$ and consider ${g_1}^{\boldsymbol{a}}_{\{i,j\}}(3,1)$. As the 2×2-plane spanned by inputs 2 and 3 in dimension i and inputs 1 and 2 in dimension j is an Aff_3 we must have ${g_1}^{\boldsymbol{a}}_{\{i,j\}}(3,1) = \alpha$. But then g_1 would have a link over α in dimension i between inputs 1 and 3 violating that g_1 is link-free. Thus, $|A_i| = 2$, and by an analogous argument $|A_j| = 2$.

We are now ready to apply Lemma 14. As the 2×2-planes spanned by dimensions i and j in g_1 and g_2 are Aff_3 with two α's, they have exactly one diagonal with two α's. For g_m with $m > 2$ we see that each 2×2-plane has at least a diagonal with two α's since g_m must have an α where g_1 has a β or a γ as to not have a link to g_1 or g_2 and each 2×2-plane in g_1 has exactly one diagonal with a β and a γ. As g_m is link-free each 2×2-plane cannot have more than one diagonal with two α's. Thus, the conditions of Lemma 14 are fulfilled and we conclude f is collapsible by collapsing β and γ. We conclude the proof of this case by displaying a function f of this form in Figure 8.

$$\begin{array}{cc|cc|cc|cc}
\alpha & \beta & \alpha & \gamma & \beta & \alpha & \gamma & \alpha \\
\gamma & \alpha & \beta & \alpha & \alpha & \gamma & \alpha & \beta
\end{array}$$

Fig. 8. A function f where all induced rectangles in g_1 are Aff_3 with two α's and where for $i > 2$, g_i does not have links to g_1 or g_2.

Case 2: g_1 with both XOR and Aff_3 (spanned by i, j). By Lemma 15 we have that f is collapsible.

Case 3: g_1 with only XOR's, one type of XOR's (spanned by i, j). We now consider the case when all 2×2-planes in g_1 spanned by dimensions i, j are XOR's, all of the same type. We assume the type is (α, β)-XOR's, which is without loss of generality as we can reorder dimensions and we know α must occur in g_1 as there is a link between g_1 and g_2 over α. As $|A_i| = |A_j| = 2$, we know from Lemma 15 that if any 2×2 plane spanned by dimensions i, j (in any g_m) is of the form Aff_3, then f is collapsible.

What remains is to analyze the situation when all 2×2-planes in f spanned by dimensions i, j in all g_m are XOR's. If none of the planes contain (β, γ)-XOR's then each plane has a diagonal with two α's and by Lemma 14 f is collapsible. As $g_2 = \pi_2 \circ g_1$ we know that all XOR's in g_2 are (α, γ)-XOR's. If for some m, g_m has an (β, γ)-XOR spanned by dimensions i, j then by Lemma 13 we know that there is a dimension k such that the input in k determines the type of XOR. As there are (α, β)-XOR's when $x_1 = 1$ and (α, γ)-XOR's when $x_1 = 2$ we see that $k = 1$ and f is a permuted sum with party 1 as the permuter.

Case 4: g_1 with only XOR's, two types of XOR's (spanned by i, j). We now proceed to the case where all 2×2-planes spanned by dimensions i, j in g_1 are XOR's, and there are two types of XOR's among them. We assume that g_1 has an (α, β)-XOR which is without loss of generality as we know that g_1 has at least one α.

We now claim that there is no (β, γ)-XOR in f spanned by dimensions i, j. Assume to the contrary that there is a (β, γ)-XOR in f. Then Lemma 13 applies and we know there is a dimension k such that the input in dimension k determines the type of XOR. Let $\boldsymbol{a}$ be inputs such that $f^{\boldsymbol{a}}_{\{1,i,j\}}(1, \cdot, \cdot)$ is an (α, β)-XOR. Then as $g_1 = \pi_2 \circ g_2$ we see that $f^{\boldsymbol{a}}_{\{1,i,j\}}(2,\cdot,\cdot)$ is an (α,γ)-XOR. As changing the input x_1 (keeping all other inputs fixed) changes the type of XOR we must have $k = 1$. But we have assumed that g_1 contains two types of XOR, so k cannot be 1 and we get a contradiction.

The only remaining possibility is that all the 2×2-planes spanned by dimensions i, j in f are (α,β)-XOR's and (α,γ)-XOR's. Thus for each 2×2-plane there is a diagonal with two α's and by Lemma 14 we know that f is collapsible by collapsing β and γ.

Case 5: g_1 with only XOR's, three types of XOR's (spanned by i, j). We claim that our final case is such that there are no functions to which it applies. In the previous case with g_1 with only XOR's and two types of XOR's we showed that there could be no (β,γ)-XOR in f. Our proof made use of the fact that there were *at least* two types of XOR's in g_1. Thus, there are no corner-free functions with links in one dimension such that g_1 consists of only XOR's with all three types of XOR's. □

A.7 Proof of Lemma 18

Lemma 18. *Every n-argument function $f : A_1 \times \ldots \times A_n \to \mathbb{Z}_3$ with links in 2 or more dimensions and without an embedded* OR *is collapsible.*

Proof. We reorder inputs and dimensions such that there are links between inputs 1 and 2 in both dimensions 1 and 2. We begin by showing that a normalized f cannot have links over two different outputs in two different dimensions. Assume to the contrary that there is a link over output α in dimension 1 for inputs x_1, x_2, and a link over output β in dimension 2 for inputs y_1, y_2.

Let π_α be the permutation transposing β and γ, written $(\alpha\ \gamma\ \beta)$, and π_β be the permutation transposing α and γ, written $(\gamma\ \beta\ \alpha)$. For each $\boldsymbol{a}$ by Corollary 10 we have that $f^{\boldsymbol{a}}_{\{1,2\}}(x_2, y_1) = \pi_\alpha\{f^{\boldsymbol{a}}_{\{1,2\}}(x_1, y_1)\}$ and $f^{\boldsymbol{a}}_{\{1,2\}}(x_2, y_2) = \pi_\beta\{f^{\boldsymbol{a}}_{\{1,2\}}(x_1, y_2)\}$. Corollary 10 also gives that $f^{\boldsymbol{a}}_{\{1,2\}}(x_1, y_2) = \pi_\beta\{f^{\boldsymbol{a}}_{\{1,2\}}(x_1, y_1)\}$ and $f^{\boldsymbol{a}}_{\{1,2\}}(x_2, y_2) = \pi_\alpha\{f^{\boldsymbol{a}}_{\{2,1\}}(x_1, y_2)\}$. However, the permutation $\pi_\alpha \circ \pi_\beta = (\beta\ \gamma\ \alpha)$ and $\pi_\beta \circ \pi_\alpha = (\gamma\ \alpha\ \beta)$, and hence for all $x : \pi_\alpha\{\pi_\beta(x)\} \neq \pi_\beta\{\pi_\alpha(x)\}$ giving a contradiction.

We relabel and reorder inputs of f such that all links in f are over output α and that there are links in both dimensions 1 and 2 with inputs 1 and 2. This is without loss of generality as the fact that the link in dimension 1 has the same inputs as the link in dimension 2 does not affect anything. We illustrate two functions with links in two dimensions with inputs 1 and 2 in Figure 9

What are the possible substructures spanned by inputs 1 and 2 in dimensions 1 and 2 in f? By Corollary 10 we see that an output in such a 2×2 substructure uniquely determines the others and we either get an (α)-constant or a (β,γ)-XOR. We illustrate the possibilities in Figure 10.

It follows from the fact that f cannot have links over outputs other than α, together with Corollary 11 that all dimensions $|A_i| \leq 4$ (in a row, there can be at most two α's, one β and one γ). We now consider the whole planes spanned

$$\begin{array}{ccc} \beta & \gamma & \alpha \\ \gamma & \beta & \alpha \\ \alpha & \alpha & \beta \end{array} \qquad \begin{array}{ccc} \alpha & \alpha & \beta \\ \alpha & \alpha & \gamma \\ \gamma & \beta & \alpha \end{array}$$

(a) f_1 with two links (b) f_2 with two links

Fig. 9. Two different functions with links in two dimensions.

$$\begin{array}{cc} \alpha & \alpha \\ \alpha & \alpha \end{array} \qquad \begin{array}{cc} \beta & \gamma \\ \gamma & \beta \end{array} \qquad \begin{array}{cc} \gamma & \beta \\ \beta & \gamma \end{array}$$

(a) (α)-constant (b) (β, γ)-XOR (c) (β, γ)-XOR

Fig. 10. The three possibilities for the 2×2-plane spanned by inputs 1 and 2 in dimensions 1 and 2.

by dimensions 1 and 2. For every plane, we know that there is a 2×2 square of either the form (α)-constant or (β, γ)-XOR spanned by inputs 1 and 2. What does this imply for the remainder of the plane?

We claim that there is no link in dimension 1 between inputs 1 and 3. This follows since there are no links in f over β or γ and since there is a link between inputs 1 and 2 in dimension α by Corollary 11 inputs 1 and 3 in dimension 1 cannot have a link over α. This means that $f^{\boldsymbol{a}}_{\{1\}}(1) = \alpha \iff f^{\boldsymbol{a}}_{\{1\}}(3) \neq \alpha$ for all $\boldsymbol{a}$. Analogously, $f^{\boldsymbol{a}}_{\{1\}}(1) = \alpha \iff f^{\boldsymbol{a}}_{\{1\}}(4) \neq \alpha$ for all $\boldsymbol{a}$. Thus, we see that $f^{\boldsymbol{a}}_{\{1\}}(3) = \alpha \iff f^{\boldsymbol{a}}_{\{1\}}(4) = \alpha$ for all $\boldsymbol{a}$. Since f has a link over α in dimension 1 and 2, by Corollary 10 we have that $f^{\boldsymbol{a}}_{\{1\}}(1) = \alpha \iff f^{\boldsymbol{a}}_{\{1\}}(4) = \alpha$ for all $\boldsymbol{a}$. An identical argument applies for dimension 2.

Recalling the notation $\not\alpha$ used to denote an output which is either β or γ, we see that there are only two possibilities for the planes spanned by dimensions 1 and 2. We illustrate these for the maximal case when $|A_1| = |A_2| = 4$ in Figure 11.

$$\begin{array}{cccc} \alpha & \alpha & \not\alpha & \not\alpha \\ \alpha & \alpha & \not\alpha & \not\alpha \\ \not\alpha & \not\alpha & \alpha & \alpha \\ \not\alpha & \not\alpha & \alpha & \alpha \end{array} \qquad \begin{array}{cccc} \not\alpha & \not\alpha & \alpha & \alpha \\ \not\alpha & \not\alpha & \alpha & \alpha \\ \alpha & \alpha & \not\alpha & \not\alpha \\ \alpha & \alpha & \not\alpha & \not\alpha \end{array}$$

Fig. 11. The two possible structures for planes spanned by dimensions 1 and 2.

We claim that f is collapsible by collapsing β and γ and define the collapsed function

$$g(\boldsymbol{x}) = \begin{cases} 1 & \text{if } f(\boldsymbol{x}) \in \{\beta, \gamma\} \\ 0 & \text{if } f(\boldsymbol{x}) = \alpha \end{cases} .$$

We show that g does not contain an embedded OR of degree 1. Then by Theorem 6 g is corner-free and from Theorem 5 we can then conclude that the collapsed function can be written as a Boolean sum.

Assume by contradiction that there is an embedded OR of degree 1 in g. We then show that there is an embedded OR in f, which is a contradiction since f is corner-free. If there is an embedded OR in g with three 0's, then that corresponds to an embedded OR in f with three α's, so we assume g has an embedded OR with three 1's.

We begin by showing that the embedded OR is not over dimension 1 or 2. By the structure of f we have that for a pair of inputs x_1, x_2 in dimension 1 we either have $f^{\boldsymbol{a}}_{\{1\}}(x_1) = \alpha \iff f^{\boldsymbol{a}}_{\{1\}}(x_2) = \alpha$ or $f^{\boldsymbol{a}}_{\{1\}}(x_1) = \alpha \iff f^{\boldsymbol{a}}_{\{1\}}(x_2) \neq \alpha$ for all $\boldsymbol{a}$. From this it follows that a 2×2-plane spanned by dimension 1 must have an even number of α's, so in particular it cannot have one α. The case for dimension 2 is analogous.

We assume there is an embedded OR in g with three 1's in dimensions j and k with inputs x_1, x_2 and y_1, y_2 and consider the planes spanned by dimensions 1 and 2. We let x_1 and y_1 be the inputs of the 0 in the embedded OR in g. We consider the case when the embedded OR occurs at input 1 in both dimensions 1 and 2, the case for when it occurs at some other input in dimensions 1 and 2 is completely analogous. Let the inputs where the embedded OR occurs be $\boldsymbol{a}$. If $|A_1| > 2$ (or $|A_2| > 2$) then since $f^{\boldsymbol{b}}_{\{1\}}(1) = \alpha \iff f^{\boldsymbol{b}}_{\{1\}}(3) \neq \alpha$ for all $\boldsymbol{b}$ we see that of the four outputs $f^{\boldsymbol{a}}_{\{1,j,k\}}(3, x_1, y_1)$, $f^{\boldsymbol{a}}_{\{1,j,k\}}(3, x_1, y_2)$, $f^{\boldsymbol{a}}_{\{1,j,k\}}(3, x_2, y_1)$, and $f^{\boldsymbol{a}}_{\{1,j,k\}}(3, x_2, y_2)$ precisely three must be α's, showing an embedded OR in f.

Finally, we let $|A_1| = |A_2| = 2$. We know that of the considered 2×2-planes spanned by dimension 1 and 2, three are of the form (β, γ)-XOR and one is (α)-constant. We claim that there is an embedded OR in f of degree 2 with $S_1 = \{1, j\}$ and $S_2 = \{2, k\}$ using inputs $(1, x_1); (2, x_2)$ on S_1 and $(1, y_1); (2, y_2)$ on S_2. We illustrate this in Figure 12, and then give a formal proof.

$$\begin{array}{cc|cc}
\alpha & \alpha & \beta & \gamma \\
\alpha & \alpha & \gamma & \beta \\
\hline
\beta & \gamma & \gamma & \beta \\
\gamma & \beta & \beta & \gamma
\end{array}$$

Fig. 12. An embedded OR in f with dimensions 1 and 2 spanning the 2×2 planes with dimension j over the horizontal line and dimension k over the vertical.

We now verify that we have an embedded OR in f as claimed. We know that $f^{\boldsymbol{a}}_{\{1,2,j,k\}}(1, 1, x_1, y_1) = \alpha$ and all of $f^{\boldsymbol{a}}_{\{1,2,j,k\}}(1, 2, x_1, y_2)$, $f^{\boldsymbol{a}}_{\{1,2,j,k\}}(2, 1, x_2, y_1)$, and $f^{\boldsymbol{a}}_{\{1,2,j,k\}}(2, 2, x_2, y_2)$ are different from α. As f has no links over outputs β or γ $f^{\boldsymbol{a}}_{\{1,2,j,k\}}(1, 2, x_1, y_2) \neq f^{\boldsymbol{a}}_{\{1,2,j,k\}}(2, 2, x_1, y_2)$ and $f^{\boldsymbol{a}}_{\{1,2,j,k\}}(2, 2, x_1, y_2) \neq f^{\boldsymbol{a}}_{\{1,2,j,k\}}(2, 2, x_2, y_2)$, which implies $f^{\boldsymbol{a}}_{\{1,2,j,k\}}(1, 2, x_1, y_2) = f^{\boldsymbol{a}}_{\{1,2,j,k\}}(2, 2, x_2, y_2)$. Analogously we see that $f^{\boldsymbol{a}}_{\{1,2,j,k\}}(2, 2, x_2, y_2) = f^{\boldsymbol{a}}_{\{1,2,j,k\}}(2, 1, x_2, y_1)$ showing that we do have an embedded OR as claimed. Thus, we conclude the proof that if f is not collapsible by collapsing β and γ, then f is not corner-free. $\square$

A.8 Proof of Lemma 21

Lemma 21. *Protocol 3 can evaluate all corner-free, collapsible functions with range $\mathbb{Z}_3$.*

Proof. Let f be corner-free and collapsible by collapsing α and β. We prove that there are functions g_i such that $f(x_1, \dots, x_n) = \alpha \implies \sum_{i=1}^{n} g_i(x_i) = 0$ and $f(x_1, \dots, x_n) = \beta \implies \sum_{i=1}^{n} g_i(x_i) = 2$ where the sum is computed modulo 4. One may wonder why the sum is not modulo 2. At the end of the proof, we show that whether the sum must be modulo 4 or if it could be modulo 2 is a property of the function f. In Figure 13 we show a function such that the sum must be modulo 4.

One viewpoint is that we are given a partially filled function table for a function g with $g(x_1, \dots, x_n) = 0$ where $f(x_1, \dots, x_n) = \alpha$, $g(x_1, \dots, x_n) = 2$ where $f(x_1, \dots, x_n) = \beta$, and a blank where $f(x_1, \dots, x_n) = \gamma$. Our proof goes by "filling in the blanks" such that the conditions of Lemma 7 are satisfied which implies that there are g_i with the desired properties. We illustrate the partially filled function table in Figure 13.

$$\begin{array}{ccc|ccc} \alpha & \beta & \gamma & \gamma & \gamma & \alpha \\ \beta & \alpha & \gamma & \gamma & \gamma & \beta \\ \gamma & \gamma & \alpha & \beta & \alpha & \gamma \end{array}$$

(a) Collapsible f

$$\begin{array}{ccc|ccc} 0 & 2 & \cdot & \cdot & \cdot & 0 \\ 2 & 0 & \cdot & \cdot & \cdot & 2 \\ \cdot & \cdot & 0 & 2 & 0 & \cdot \end{array}$$

(b) Partial g

$$\begin{array}{ccc|ccc} 0 & 2 & 1 & 3 & 1 & 0 \\ 2 & 0 & 3 & 1 & 3 & 2 \\ 3 & 1 & 0 & 2 & 0 & 3 \end{array}$$

(c) g

Fig. 13. An example collapsible f, the corresponding partially filled function table g, and a completely defined g satisfying the conditions of Lemma 7.

We refer to the values defined in the partially filled g as *given*. The condition for Lemma 7 to apply is that all induced rectangles of g satisfy (1). Our proof strategy is to describe a procedure by which we fill in the blanks and then prove that this results in (1) holding for all induced rectangles in g by case analysis. We assign terms of the form x, $x + 2$, $-x$, and $-x + 2$ to the blanks and prove that for at least one of $x = 0$ and $x = 1$ the conditions of Lemma 7 are satisfied. The case $x = 0$ corresponds to when the g_i could have been Boolean, and $x = 1$ to when we need the full range of $\mathbb{Z}_4$.

We begin by observing that if g is completely specified (i.e., the output γ never occurs in f) then g is a boolean corner-free function and by Theorem 5 we can find $g'_i(x_i)$ such that $g(x_1, \dots, x_n) = \sum_{i=1}^{n} 2g_i(x_i)$ where the computation is modulo 4. For the degenerate case when g is given as only blanks (i.e., $f(x_1, \dots, x_n) = \gamma$ for all inputs) then we can simply let g be constant 0. Thus, we proceed with the non-degenerate case when g contains both given values and blanks.

We claim that there is no link in f over output α or β, implying that there are no links in the partially filled g over output 0 or 2. Assume to the contrary that there is a link over α (the case with a link over β is analogous) in dimension k for inputs x, y with other inputs as $\boldsymbol{a}$. Consider the pair of values $f^{\boldsymbol{b}}_{\{k\}}(x), f^{\boldsymbol{b}}_{\{k\}}(y)$ for some $\boldsymbol{b} \neq \boldsymbol{a}$. Since f is normalized it follows from Lemma 9 they cannot be (β, β) or (γ, γ). Since f is corner-free, the pair cannot contain exactly one α. Furthermore, as f is collapsible by collapsing α and β the pair cannot contain exactly one β either, as then the collapsed function has an embedded OR. Thus, the only remaining option is (α, α), and this must hold for all $\boldsymbol{b} \neq \boldsymbol{a}$, showing inputs x and y are redundant in dimension k which contradicts that f is normalized.

As f does not have any links over α or β and by Corollary 11 can have at most one link over γ in a dimension, we see that for all $i, |A_i| \leq 4$ (at most one α, one β and two γ). This means that for each row in dimension 1 (a subfunction on the form $g^{\boldsymbol{a}}_{\{1\}}$ for fixed $\boldsymbol{a}$) the function g can have at most one 0, one 2, and two blanks.

Consider the "first" row, by which we mean $g^{\mathbf{1}}_{\{1\}}$ where $\mathbf{1} = (1, 1, \ldots, 1)$. Without loss of generality we reorder dimensions and inputs such that $g^{\mathbf{1}}_{\{1\}}(1)$ is given and $g^{\mathbf{1}}_{\{1\}}(2)$ is blank. We let $d_1 = g^{\mathbf{1}}_{\{1\}}(1)$. We fill in the blank at $g^{\mathbf{1}}_{\{1\}}(2)$ with x, and the second blank (if there is one) with $x + 2$.

We proceed to make an observation on the pattern of blank outputs in g. As f is collapsible by collapsing α and β, by the definition of collapsible we have that there exist functions $f_i(x_i)$ such that $\sum_{i=1}^{n} f_i(x_i) = 0 \implies f(x_1, \ldots, x_n) = \gamma \implies g(x_1, \ldots, x_n)$ is blank. In particular, this implies that for all rows in g, there are only two possible patterns for where the blanks are, and the two patterns are complementary. For every dimension i (we already did this for dimension 1) such that f_i is not constant, we reorder the inputs such that $f_i(1) \neq f_i(2)$ (which means that the pattern of blanks changes between input 1 and 2).

We proceed to fill in g. For each row in dimension 1, we apply (1) as if it was at Hamming distance 1 from the first row (even though most rows are not). In more detail, for a row with the same pattern of blanks as the first row, denote the given value at $x_1 = 1$ by c_1. A blank position for $x_1 = i$ is filled with the value $c_1 - d_1 + d_i$ where d_i is value we filled in at $x_1 = i$ on the first row (and thus, either $d_i = x$ or $d_i = x + 2$). On a row with a pattern opposite of that of the first row, we fill in a blank at $x_1 = i$ with $d_i + c_2 - x$, where c_2 is the value given at $x_1 = 2$ of that row (recall that the value at $x_1 = 2$ on the first row is x). As all arithmetic is modulo 4, we make use of the equality $2 \equiv -2$, and thus the terms we fill in blanks with are: $x, x + 2, -x$, and $-x + 2$, as previously claimed. We show how g from Figure 13 is filled in in Figure 14 (but we have not reordered inputs such that the second value on the first row is a blank).

We now proceed to show that the conditions of Lemma 7 are fulfilled by either $x = 0$ or $x = 1$. Following our observation on the pattern of blanks, we see that every induced rectangle in g must have an even number of terms with x's. We now proceed with a case analysis for each induced rectangle in g, with five cases. We prove that for the first four of the cases, no condition is imposed

$$\begin{array}{ccc|ccc}
0 & 2 & x & -x & (-x+2) & 0 \\
2 & 0 & (x+2) & (-x+2) & -x & 2 \\
-x & (-x+2) & 0 & 2 & 0 & (x+2)
\end{array}$$

Fig. 14. Filling in f from Figure 13.

upon x. The presence of rectangles in the final case determines if $x = 0$ or $x = 1$ fulfills the conditions of Lemma 7, and if no such rectangle is present in g, the both choices for x work. We remark that our construction is such that if $x = y$ satisfies the conditions of Lemma 7, then so does $x = y + 2$, however, we do not formally prove this. We also remark that for $x = 0$ all blanks will be assigned the value 0 or 2, and for $x = 1$ all blanks will be assigned the value 1 or 3.

Our five cases for an induced rectangle in g are:

1. Four given values
2. Four terms with x's
3. Two terms with x's which are neighbors
4. Two terms with x's on a diagonal, opposite signs on the x terms
5. Two terms with x's on a diagonal, same signs on the x terms

Case 1: Four given values. First, consider an induced rectangle consisting only of values 0 and 2. The condition that f is corner-free implies that (1) is satisfied for such a rectangle.

Case 2: Four terms with x's. Next, consider an induced rectangle consisting only of terms involving x's. Due to how we assigned the terms, two terms with x's which are neighbors (Hamming distance 1) must have the same sign, and precisely one of them must have a $+2$ term. Thus, in an induced rectangle, two of the terms contain a $+2$ term, (1) becomes one of $x + 2 + x + 2 \equiv x + x$, or $-x + 2 - x + 2 \equiv -x - x$, both of which are tautologies.

Case 3: Two terms with x's which are neighbors. As the third case, consider an induced rectangle with two terms involving x's, such that the two terms involving x's are neighbors. Then, by the fact that there are no links over outputs 0 and 2, and that neighboring x's have the same sign and differ by 2, we see that the equation must be one of $x+2+2 \equiv x+0$, $x+2+0 \equiv x+2$, $-x+2+2 \equiv -x+0$, and $-x + 2 + 0 \equiv -x + 2$, which are all tautologies.

Case 4: Two terms with x's on a diagonal, opposite signs on the x terms. In our fourth case, we consider an induced rectangle with two non-neighboring terms involving x where the two x's have opposite signs. We note that, for a fixed input to x_1, all x's have the same sign, so such a rectangle must involve the dimension 1. Let one of the rows contain the terms $x + c_1$ and c_2, and the second contain c_3, and $-x + c_4$, with c_1, c_2, c_3, c_4 being constants. Let $x + d_1$ be the value in the first row at the same position as the $x + c_1$ and c_3 terms, and let d_2 be the value in the first row at the same position as the c_2 and $-x + c_4$ terms. We now see from the procedure used to assign the x terms that $c_1 \equiv d_1 + c_2 - d_2$, and $c_4 \equiv d_2 + c_3 - d_1$. Thus, $x + c_1 - x + c_4 \equiv d_1 + c_2 - d_2 + d_2 + c_3 - d_1 \equiv c_2 + c_3$, as required to satisfy (1).

Case 5: Two terms with x's on a diagonal, same signs on the x terms. Finally, we proceed to the last, and most complicated, case, with an induced rectangle with two non-neighboring terms involving x, such that both x's have the same sign. For this case, we'll need the following observation:

For every dimension j, there is a pair of constants, $p_{j,1}$ and $p_{j,2} \in 0, 2$ such that, for every $\boldsymbol{a}$ such that $g^{\boldsymbol{a}}_{\{1,j\}}(1,1), g^{\boldsymbol{a}}_{\{1,j\}}(2,2) \in \{0,2\}$:

$$g^{\boldsymbol{a}}_{\{1,j\}}(1,1) = g^{\boldsymbol{a}}_{\{1,j\}}(2,2) + p_{j,1}\,, \tag{3}$$

and for every $\boldsymbol{a}$ such that $g^{\boldsymbol{a}}_{\{1,j\}}(2,1), g^{\boldsymbol{a}}_{\{1,j\}}(1,2) \in \{0,2\}$:

$$g^{\boldsymbol{a}}_{\{1,j\}}(2,1) = g^{\boldsymbol{a}}_{\{1,j\}}(1,2) + p_{j,2}\,. \tag{4}$$

The observation follows from the fact that f does not contain an embedded OR. This, since, if there are two vectors of inputs $\boldsymbol{a}$, $\boldsymbol{b}$ such that

$$\begin{array}{c} g^{\boldsymbol{a}}_{\{1,j\}}(1,1) - g^{\boldsymbol{a}}_{\{1,j\}}(2,2) \neq \\ g^{\boldsymbol{b}}_{\{1,j\}}(1,1) - g^{\boldsymbol{b}}_{\{1,j\}}(2,2) \end{array},$$

then an odd number of the above values must be 0, and the remaining must be 2. Thus, f has an embedded OR with partition $S_1 = \{i,j\}$ and $S_2 = S_1^C$ using inputs $(1,1);(2,2)$ and $\boldsymbol{a};\boldsymbol{b}$. The case for $p_{j,2}$ is analogous.

Furthermore, we claim that for any pair of dimensions j,k, it must be the case that $p_{j,1} + p_{j,2} + p_{k,1} + p_{k,2} \equiv 0 \pmod 4$, and thus either for all j we have $p_{j,1} + p_{j,2} \equiv 0$, or for all j, $p_{j,1} + p_{j,2} \equiv 2$. Fix a pair of dimensions j,k and inputs $\boldsymbol{a}$. We begin by noting that if j is such that f_j is a constant function, and thus the pattern of blanks does not depend on x_j, then $p_{j,1}$ and $p_{j,2}$ are undefined, and analogously for k. We reordered inputs such that $f_1(1) \neq f_1(2)$, implying that exactly one of $f^{\boldsymbol{a}}_{\{1,j,k\}}(1,1,1)$ and $f^{\boldsymbol{a}}_{\{j,k\}}(2,1,1)$ is α or β and thus that the value on one of those positions in g is given.

Consider the case when $g^{\boldsymbol{a}}_{\{1,j,k\}}(1,1,1)$ is given (the case when $f^{\boldsymbol{a}}_{\{1,j,k\}}(2,1,1)$ is given is analogous) and let $c = g^{\boldsymbol{a}}_{\{1,j,k\}}(1,1,1)$. Since $f_1(1) \neq f_1(2)$ and $f_j(1) \neq f_j(2)$, we claim that $g^{\boldsymbol{a}}_{\{1,j,k\}}(2,2,1)$ is given. This follows since $f_1(1) + f_j(1) + f_k(1) \equiv f_1(2) + f_j(2) + f_k(1) \pmod 2$. Then by (3) we see that $c \equiv g^{\boldsymbol{a}}_{\{1,j,k\}}(2,2,1) + p_{j,1}$. Analogously in dimension k, (4) implies $g^{\boldsymbol{a}}_{\{1,j,k\}}(2,2,1) \equiv g^{\boldsymbol{a}}_{\{1,j,k\}}(1,2,2) + p_{k,2}$ and thus $c \equiv g^{\boldsymbol{a}}_{\{1,j,k\}}(1,2,2) + p_{k,2} + p_{j,1}$. Continuing, (4) gives $g^{\boldsymbol{a}}_{\{1,j,k\}}(2,1,2) \equiv g^{\boldsymbol{a}}_{\{1,j,k\}}(1,2,2) + p_{j,2}$. As all values involved are 0 or 2 and we are working modulo 4, we can apply the equivalence $y \equiv -y$ and rewrite as $g^{\boldsymbol{a}}_{\{1,j,k\}}(1,2,2) \equiv g^{\boldsymbol{a}}_{\{1,j,k\}}(2,1,2) + p_{j,2}$ so $c \equiv g^{\boldsymbol{a}}_{\{1,j,k\}}(2,1,2) + p_{j,2} + p_{k,2} + p_{j,1}$. And, finally, by (3) we have $g^{\boldsymbol{a}}_{\{1,j,k\}}(1,1,1) \equiv g^{\boldsymbol{a}}_{\{1,j,k\}}(1,2,2) + p_{k,1}$ which after rewriting results in $c \equiv g^{\boldsymbol{a}}_{\{1,j,k\}}(1,1,1) + p_{j,1} + p_{j,2} + p_{k,1} + p_{k,2}$. Since we defined $c = g^{\boldsymbol{a}}_{\{1,j,k\}}(1,1,1)$ we see that $p_{j,1} + p_{j,2} + p_{k,1} + p_{k,2} \equiv 0$. We illustrate the proof in Figure 15.

With these observations, we are ready to complete the discussion of the final case of this proof. Recall that we reordered inputs such that the first two positions of the first row are d_1, and x.

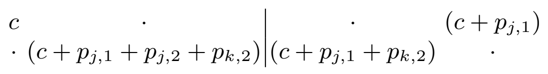


Fig. 15. Proving that $p_{j,1}+p_{j,2}+p_{k,1}+p_{k,2}\equiv 0$. Dimensions 1 and k span the 2×2 planes.

By our construction, a rectangle with two terms involving x's with the same sign on a diagonal occurs only when the input x_1 is fixed. Furthermore, we see that if the rectangle considered is spanned by dimensions i, j with inputs a_1, a_2 in dimension i and b_1, b_2 in dimension j then $f_i(a_1)\neq f_i(a_2)$ and $f_j(b_1)\neq f_j(b_2)$, as otherwise the rectangle does not have a diagonal with terms involving x's.

We further claim that we can restrict our attention to $x_i\in\{1,2\}$ for all i. Recall that we reordered inputs such that for all dimensions affecting the pattern of blanks it changes between input 1 and 2. By the fact that f has no links over output α and β there are no links over given values 0 or 2 in g. This, together with our construction implies that for every dimension k, if a_1, a_2 are inputs such that $f_k(a_1)=f_k(a_2)$ (or, equivalently, $g^{a_1}_{\{k\}}$ has the same pattern of blanks as $g^{a_2}_{\{k\}}$) we have for all inputs $\boldsymbol{b}$ that $g^{\boldsymbol{b}}_{\{k\}}(a_1)\equiv g^{\boldsymbol{b}}_{\{k\}}(a_2)+2 \pmod 4$. Considering the structure of (1) we see that it holds for $x_k=a_1$ iff it holds for $x_k=a_2$.

Recall that we reordered inputs such that $d_1=g(\mathbf{1})$ is given. We now consider an induced rectangle spanned by inputs 1 and 2 in both dimensions j and k with $x_1=1$ with others inputs $\boldsymbol{a}$. We have $g^{\boldsymbol{a}}_{\{1,j,k\}}(1,1,1)=-x+c_1$, $g^{\boldsymbol{a}}_{\{1,j,k\}}(1,1,2)=c_2$, $g^{\boldsymbol{a}}_{\{1,j,k\}}(1,2,1)=c_3$, and $g^{\boldsymbol{a}}_{\{1,j,k\}}(1,2,2)=-x+c_4$. We know that $g^{\boldsymbol{a}}_{\{1,j,k\}}(2,1,1)$ must be given, and we denote it by c_5. By our construction, $c_1=d_1+c_5$ and reordering, we see that $c_5=c_1-d_1$. Analogously $g^{\boldsymbol{a}}_{\{1,j,k\}}(2,2,2)=c_4-d_1$. Furthermore, by (4) we have that $c_1-d_1\equiv c_2+p_{j,2}$ and by (3) we have that $c_3\equiv c_4-d_1+p_{j,1}$. As $c_1,c_2,c_3,c_4,d_1,p_{j,1},p_{j,2}\in\{0,2\}$ we can use the identities $y\equiv -y$ and $2y\equiv 0$ with them. We need to show $-x+c_1-x+c_4\equiv c_2+c_3$. Rewriting gives $2x\equiv c_1+c_2+c_3+c_4\equiv 2c_2+2c_3+2d_1+p_{j,1}+p_{j,2}\equiv p_{j,1}+p_{j,2}$. As $p_{j,1}+p_{j,2}$ is a constant depending on f which is either 0 or 2, we see that $x=0$ or $x=1$ satisfies all equations on this form.

The case for $x_1=2$ is almost identical. Consider an induced rectangle spanned by inputs 1 and 2 in both dimensions j and k with $x_1=2$ and others inputs as $\boldsymbol{a}$. We have $g^{\boldsymbol{a}}_{\{1,j,k\}}(2,1,1)=x+c_1$, $g^{\boldsymbol{a}}_{\{1,j,k\}}(2,1,2)=c_2$, $g^{\boldsymbol{a}}_{\{1,j,k\}}(2,2,1)=c_3$, and $g^{\boldsymbol{a}}_{\{1,j,k\}}(2,2,2)=x+c_4$. We know that $g^{\boldsymbol{a}}_{\{1,j,k\}}(2,1,1)$ must be given, and we denote it by c_5. We have that $c_1\equiv c_5-d_1$ by our construction, so $c_5\equiv c_1+d_1$. Analogously, $g^{\boldsymbol{a}}_{\{1,j,k\}}(2,2,1)=c_4+d_1$. Furthermore, by (3) we have that $c_1+d_1\equiv c_2+p_{j,1}$ and by (4) we have that $c_3\equiv c_4+d_1+p_{j,2}$. As $c_1,c_2,c_3,c_4,d_1,p_{j,1},p_{j,2}\in\{0,2\}$ we can use the identities $y\equiv -y$ and $2y\equiv 0$ with them. We need to show $x+c_1+x+c_4\equiv c_2+c_3$. Rewriting gives $2x\equiv c_1+c_2+c_3+c_4\equiv 2c_2+2c_3+2d_1+p_{j,1}+p_{j,2}\equiv p_{j,1}+p_{j,2}$. As $p_{j,1}+p_{j,2}$ is a constant depending on f which is either 0 or 2, we see that $x=0$ or $x=1$ satisfies all equations on this form.

□

A.9 Proof of Theorem 22

Theorem 22. *Every n-argument function $f : A_1 \times \ldots A_n \to \mathbb{Z}_3$ that has an embedded* OR *of any degree has an embedded* OR *of degree at most* 3*. Furthermore, every* 4*-argument function $f : A_1 \times A_2 \times A_3 \times A_4 \to \mathbb{Z}_3$ that has an embedded* OR*, also has one of degree at most* 2*.*

Proof. Let f have an embedded OR of degree $k \geq 3$, corresponding to a partition X_1, X_2, A, with $|X_1| \geq |X_2|$. We reorder inputs such that $X_1 = \{x_1, x_2, \ldots, x_k\}$. We then show that we can find a new partition X_1', X_2', A', either such that $|X_1'| < |X_1|$ and $X_2 = X_2'$, or such that $|X_1'| \leq 2$, and $|X_2'| \leq |X_2| + 1$, also corresponding to an embedded OR. From this, the theorem follows.

We relabel outputs such that the embedded OR contains three α's and one β and let $\boldsymbol{a}, \boldsymbol{b}$ be inputs for X_1, and $\boldsymbol{c}, \boldsymbol{d}$ for X_2 realizing the embedded OR. Thus, there exists $\boldsymbol{e}$ such that $f^{\boldsymbol{e}}_{\{X_1,X_2\}}(\boldsymbol{a}, \boldsymbol{c}) = f^{\boldsymbol{e}}_{\{X_1,X_2\}}(\boldsymbol{b}, \boldsymbol{c}) = f^{\boldsymbol{e}}_{\{X_1,X_2\}}(\boldsymbol{a}, \boldsymbol{d}) = \alpha$ and $f^{\boldsymbol{e}}_{\{X_1,X_2\}}(\boldsymbol{b}, \boldsymbol{d}) = \beta$. For $i \leq k$, let a_i be the element in $\boldsymbol{a}$ that is used for input x_i, and b_i the corresponding element in $\boldsymbol{b}$. We write a vector with a single element x as (x).

We now consider what happens if we take $\boldsymbol{a}$ and replace some values with the corresponding values from from $\boldsymbol{b}$. To simplify the discussion, we define for $S \subseteq X_1$ the function $g(S, \boldsymbol{x}) = f^{\boldsymbol{e}}_{\{X_1,X_2\}}(\boldsymbol{a}_{S=\boldsymbol{b}}, \boldsymbol{x})$ where $\boldsymbol{a}_{S=\boldsymbol{b}}$ denotes $\boldsymbol{a}$ with values as indicated by the subset S replaced by the corresponding values from $\boldsymbol{b}$. We proceed with a case analysis depending on what values $g(S, \boldsymbol{c})$ and $g(S, \boldsymbol{d})$ takes for nonempty $S \subset X_1$. In each case, we assume the previously discussed cases do not apply. For instance, in case 3 we assume that there is no S such that $g(S, \boldsymbol{c}) = \alpha$. We proceed with the following cases:

1. $g(S, \boldsymbol{c}) = \alpha$ or $g(S, \boldsymbol{d}) = \alpha$ for some S
2. $g(S, \boldsymbol{c}) = \beta$ and $g(S, \boldsymbol{d}) = \beta$ for some S
3. $g(S, \boldsymbol{c}) = \gamma$ and $g(S, \boldsymbol{d}) = \gamma$ for some S
4. The pair $g(S, \boldsymbol{c}), g(S, \boldsymbol{d}) \in \{(\beta, \gamma), (\gamma, \beta)\}$ for all S

$\mathsf{Case\ 1}$: $g(S, \boldsymbol{c}) = \alpha$ or $g(S, \boldsymbol{d}) = \alpha$ for some S. If $g(S, \boldsymbol{c}) \neq g(S, \boldsymbol{d})$ then we claim that there is an embedded OR with $X_1' = S$ and $X_2' = X_2$. Assume $g(S, \boldsymbol{c}) = \alpha$ and let $\boldsymbol{e}'$ be $\boldsymbol{e}$ extended with the elements from $\boldsymbol{a}$ for arguments in $X_1' \backslash X_1$, let $\boldsymbol{a}'$ be the elements of $\boldsymbol{a}$ in $X_1' \cap X_1$, and let $\boldsymbol{b}'$ be the elements of $\boldsymbol{b}$ in $X_1' \cap X_1$. Then we can verify that

$$\begin{aligned}
f^{\boldsymbol{e}'}_{\{X_1',X_2'\}}(\boldsymbol{a}', \boldsymbol{c}) &= f^{\boldsymbol{e}}_{\{X_1,X_2\}}(\boldsymbol{a}, \boldsymbol{c}) && = \alpha \\
f^{\boldsymbol{e}'}_{\{X_1',X_2'\}}(\boldsymbol{a}', \boldsymbol{d}) &= f^{\boldsymbol{e}}_{\{X_1,X_2\}}(\boldsymbol{a}, \boldsymbol{d}) && = \alpha \\
f^{\boldsymbol{e}'}_{\{X_1',X_2'\}}(\boldsymbol{b}', \boldsymbol{c}) &= g(S, \boldsymbol{c}) && = \alpha \\
f^{\boldsymbol{e}'}_{\{X_1',X_2'\}}(\boldsymbol{b}', \boldsymbol{d}) &= g(S, \boldsymbol{d}) && \neq \alpha\,.
\end{aligned}$$

If $g(S, \boldsymbol{c}) = g(S, \boldsymbol{d}) = \alpha$ then we claim there is an embedded OR with $X_1' = X_1 \backslash S$ and $X_2' = X_2$. Let $\boldsymbol{e}'$ be $\boldsymbol{e}$ extended with the elements from $\boldsymbol{b}$ for arguments in $X_1 \backslash X_1'$, let $\boldsymbol{a}'$ be the elements of $\boldsymbol{a}$ in $X_1' \cap X_1$, and let $\boldsymbol{b}'$ be the elements of $\boldsymbol{b}$ in $X_1' \cap X_1$. Then we can verify that

$$\begin{aligned}
f^{\boldsymbol{e}'}_{\{X_1',X_2'\}}(\boldsymbol{a}',\boldsymbol{c}) &= g(S,\boldsymbol{c}) &&= \alpha \\
f^{\boldsymbol{e}'}_{\{X_1',X_2'\}}(\boldsymbol{a}',\boldsymbol{d}) &= g(S,\boldsymbol{d}) &&= \alpha \\
f^{\boldsymbol{e}'}_{\{X_1',X_2'\}}(\boldsymbol{b}',\boldsymbol{c}) &= f^{\boldsymbol{e}}_{\{X_1,X_2\}}(\boldsymbol{b},\boldsymbol{c}) &&= \alpha \\
f^{\boldsymbol{e}'}_{\{X_1',X_2'\}}(\boldsymbol{b}',\boldsymbol{d}) &= f^{\boldsymbol{e}}_{\{X_1,X_2\}}(\boldsymbol{b},\boldsymbol{d}) &&= \beta\,.
\end{aligned}$$

Case 2: $g(S,\boldsymbol{c}) = \beta$ and $g(S,\boldsymbol{d}) = \beta$ for some S. This case is analogous to the case when $g(S,\boldsymbol{c}) = g(S,\boldsymbol{d}) = \alpha$.

Case 3: $g(S,\boldsymbol{c}) = \gamma$ and $g(S,\boldsymbol{d}) = \gamma$ for some S. We assume the previous cases did not apply to g, and thus that for all non-empty $S' \subset X_1$ we have that the pair $g(S',\boldsymbol{c}), g(S',\boldsymbol{d}) \in \{(\beta,\gamma),(\gamma,\beta),(\gamma,\gamma)\}$. We begin with the case that for all non-empty $S' \subset X_1$ we have that $g(S',\boldsymbol{c}) = g(S',\boldsymbol{d}) = \gamma$. Then we claim there is an embedded OR with $X_1' = \{x_1\}$ and $X_2' = X_2 \cup \{x_2\}$. Let $\boldsymbol{e}'$ be $\boldsymbol{e}$ extended with the elements from $\boldsymbol{a}$ for arguments in $X_1 \backslash \{x_1, x_2\}$, let $\boldsymbol{a}' = (a_1)$ and $\boldsymbol{b}' = (b_1)$, let $\boldsymbol{c}'$ be $\boldsymbol{c}$ extended with a_j, and let $\boldsymbol{d}'$ be $\boldsymbol{d}$ extended with b_j. Then $f^{\boldsymbol{e}'}_{\{X_1',X_2'\}}(\boldsymbol{a}',\boldsymbol{c}') = f^{\boldsymbol{e}}_{\{X_1,X_2\}}(\boldsymbol{a},\boldsymbol{c}) = \alpha$ and $f^{\boldsymbol{e}'}_{\{X_1',X_2'\}}(\boldsymbol{a}',\boldsymbol{d}') = f^{\boldsymbol{e}'}_{\{X_1',X_2'\}}(\boldsymbol{b}',\boldsymbol{c}') = f^{\boldsymbol{e}'}_{\{X_1',X_2'\}}(\boldsymbol{b}',\boldsymbol{d}') = \gamma$. We illustrate this case in Figure 16.

$S =$	$\boldsymbol{x} = \boldsymbol{c}$	$\boldsymbol{x} = \boldsymbol{d}$
$\emptyset$	$\boldsymbol{\alpha}$	α
$\{x_1\}$	γ	$\boldsymbol{\gamma}$
$\{x_2\}$	$\boldsymbol{\gamma}$	γ
$\{x_1, x_2\}$	γ	$\boldsymbol{\gamma}$

Fig. 16. The function $g(S,\boldsymbol{x})$ when for all $S' : g(S',\boldsymbol{c}) = g(S',\boldsymbol{d}) = \gamma$ with an embedded OR marked by bold symbols.

We now consider the case when for some S', the pair $g(S',\boldsymbol{c}), g(S',\boldsymbol{d}) \in \{(\beta,\gamma),(\gamma,\beta)\}$. Then there is $S_1, S_2 \subset X_1$ differing in exactly one element (which we denote x_i) such that $g(S_1,\boldsymbol{c}) = g(S_1,\boldsymbol{d}) = \gamma$ and the pair $g(S_2,\boldsymbol{c}), g(S_2,\boldsymbol{d}) \in \{(\beta,\gamma),(\gamma,\beta)\}$. From this we see that there is an embedded OR with $X_1' = \{x_i\}$ and $X_2' = X_2$.

Case 4: The pair $g(S,\boldsymbol{c}), g(S,\boldsymbol{d}) \in \{(\beta,\gamma),(\gamma,\beta)\}$ for all S.

This case is the most complicated, and we proceed with two sub-cases. In our analysis, we define the function

$$h(y_1,\ldots,y_k) = \begin{cases} 0 & \text{if } g(S,\boldsymbol{d}) = \beta \\ 1 & \text{if } g(S,\boldsymbol{d}) = \gamma \end{cases}$$

where $S = \{x_i : y_i = 1\}$. We remark that $h(0,\ldots,0)$ is left undefined (we define its value later) and $h(1,\ldots,1) = 1$ as $g(X_1,\boldsymbol{d}) = \beta$. We claim that if there is an embedded OR in h, it corresponds to an embedded OR in f. As h is a Boolean function, we know from Theorem 6 that if there is an embedded OR in h, then there is one of degree 1.

Consider an embedded OR in h between inputs y_i and y_j. Then we know that there is an embedded OR with β and γ with corners at $f^{\boldsymbol{e}'}_{\{i,j\}}(a_i, a_j)$, $f^{\boldsymbol{e}'}_{\{i,j\}}(a_i, b_j)$, $f^{\boldsymbol{e}'}_{\{i,j\}}(b_i, a_j)$, and $f^{\boldsymbol{e}'}_{\{i,j\}}(b_i, a_j)$ where $\boldsymbol{e}'$ is $\boldsymbol{e}$ extended with the values of $\boldsymbol{d}$ on S_2, the values of $\boldsymbol{a}$ for $\{x_i : y_i = 0\}$ and the values of $\boldsymbol{b}$ for $\{x_i : y_i = 1\}$ in the embedded OR.

We now return to $h(0, \dots, 0)$ and define it to either of 0 and 1 that results in f remaining corner-free. We analyze the case when this fails and both $h(0, \dots, 0) = 0$ and $h(0, \dots, 0) = 1$ creates an embedded OR (but h was corner-free with $h(0, \dots, 0)$ undefined) and claim that there is then a small embedded OR in f.

To see this, we consider what values h takes when exactly one of its inputs is non-zero, or, equivalently, the values of $g(S, \boldsymbol{d})$ for singleton sets S which is more notationally convenient to discuss. We proceed in two cases depending on whether $g(S, \boldsymbol{d})$ has the same output for all singleton sets or not. The embedded OR prove exists turn out to be identical to those which we prove to exist in the case when h is independent of one or more inputs.

Consider the case when there is $x_i, x_j \in X_1$ such that $g(\{x_i\}, \boldsymbol{d}) \neq g(\{x_j\}, \boldsymbol{d})$. Choose x_i such that $g(\{x_i\}, \boldsymbol{d}) = g(\{x_i, x_j\}, \boldsymbol{d})$. Then we claim there is an embedded OR with $X_1' = \{x_i\}$ and $X_2' = X_2 \cup \{x_j\}$. For $\boldsymbol{a}' = (a_i)$, $\boldsymbol{b}' = (b_i)$, $\boldsymbol{c}'$ as $\boldsymbol{c}$ extended with a_j, $\boldsymbol{d}'$ as $\boldsymbol{d}$ extended with b_j and $\boldsymbol{e}'$ as $\boldsymbol{e}$ extended with values from $\boldsymbol{a}$ where $X_1 \backslash \{x_i, x_j\}$. Then we have $f^{\boldsymbol{e}'}_{\{X_1', X_2'\}}(\boldsymbol{a}', \boldsymbol{c}') = \alpha$. $f^{\boldsymbol{e}'}_{\{X_1', X_2'\}}(\boldsymbol{a}', \boldsymbol{d}') = f^{\boldsymbol{e}'}_{\{X_1', X_2'\}}(\boldsymbol{b}', \boldsymbol{c}') = f^{\boldsymbol{e}'}_{\{X_1', X_2'\}}(\boldsymbol{b}', \boldsymbol{d}') \neq \alpha$. We illustrate this in Figure 17(a).

Consider the case when for all $x_i, x_j \in X_1$ we have $g(\{x_i\}, \boldsymbol{d}) = g(\{x_j\}, \boldsymbol{d})$. Then the fact that both assigning $h(0, \dots, 0) = 0$ and assigning $h(0, \dots, 0) = 1$ implies that there is a pair x_i, x_j such that $g(\{x_i, x_j\}, \boldsymbol{d}) = g(\{x_i\}, \boldsymbol{d})$, and that there is a pair x_k, x_l such that $g(\{x_k, x_l\}, \boldsymbol{d}) \neq g(\{x_k\}, \boldsymbol{d})$. We claim that there is an embedded OR with $X_1' = \{x_i\}$ and $X_2' = \{x_j\}$. Let $\boldsymbol{e}$ be $\boldsymbol{e}$ extended with all elements from $\boldsymbol{d}$ and elements from $\boldsymbol{a}$ where $X_1 \backslash \{x_i, x_j\}$. We can verify that $f^{\boldsymbol{e}'}_{\{i,j\}}(a_i, a_j) = \alpha$ and $f^{\boldsymbol{e}'}_{\{i,j\}}(b_i, a_j) = f^{\boldsymbol{e}'}_{\{i,j\}}(a_i, b_j) = f^{\boldsymbol{e}'}_{\{i,j\}}(b_i, b_j) \neq \alpha$. We illustrate this in Figure 17(b).

If h is corner-free, then Theorem 5 gives that $h(y_1, \dots, y_k) = \sum_{i=1}^{k} f_k(y_k)$ modulo 2. We proceed in two cases, depending on whether there is an i such that h is independent of y_i, or if h depends on all its inputs.

Case 4.1: There is an i such that h is independent of y_i.

If h is independent of two variables y_i, y_j then we claim there is an embedded OR with $X_1' = \{x_i\}$ and $X_2' = \{x_j\}$. We can verify that $f^{\boldsymbol{e}'}_{\{i,j\}}(a_i, a_j) = \alpha$ and $f^{\boldsymbol{e}'}_{\{i,j\}}(a_i, b_j) = f^{\boldsymbol{e}'}_{\{i,j\}}(b_i, a_j) = f^{\boldsymbol{e}'}_{\{i,j\}}(b_i, b_j) \neq \alpha$ for $\boldsymbol{e}'$ as $\boldsymbol{e}$ extended with the values from $\boldsymbol{c}$ on X_2 and the values from $\boldsymbol{a}$ on $X_1 \backslash \{x_i, x_j\}$.

If h is independent of one variable y_i then we claim there is an embedded OR with $X_1' = \{x_i\}$ and $X_2' = X_2 \cup \{x_j\}$. We can verify that $f^{\boldsymbol{e}'}_{\{i,X_2'\}}(a_i, \boldsymbol{c}') = \alpha$ and

$S =$	$\boldsymbol{x} = \boldsymbol{c}$	$\boldsymbol{x} = \boldsymbol{d}$
$\emptyset$	$\boldsymbol{\alpha}$	α
$\{x_i\}$	β	$\boldsymbol{\gamma}$
$\{x_j\}$	$\boldsymbol{\gamma}$	β
$\{x_i, x_j\}$	β	$\boldsymbol{\gamma}$

(a) f_1

$S =$	$\boldsymbol{x} = \boldsymbol{c}$	$\boldsymbol{x} = \boldsymbol{d}$
$\emptyset$	$\boldsymbol{\alpha}$	α
$\{x_i\}$	$\boldsymbol{\beta}$	γ
$\{x_j\}$	$\boldsymbol{\beta}$	γ
$\{x_i, x_j\}$	$\boldsymbol{\beta}$	γ

(b) f_2

Fig. 17. The two cases when assigning any value to $h(0, \ldots, 0)$ creates an embedded OR in h. Identical to the cases when h is independent on one or more inputs.

$f^{\boldsymbol{e}'}_{\{i, X'_2 j\}}(a_i, \boldsymbol{d}') = f^{\boldsymbol{e}'}_{\{i, X'_2\}}(b_i, \boldsymbol{c}') = f^{\boldsymbol{e}'}_{\{i, X'_2 j\}}(b_i, \boldsymbol{d}') \neq \alpha$ for $\boldsymbol{e}'$ as $\boldsymbol{e}$ extended with the values from $\boldsymbol{a}$ on $X_1 \backslash \{x_i\}$, with $\boldsymbol{c}'$ as $\boldsymbol{c}$ extended with a_j on x_j, and with $\boldsymbol{d}'$ as $\boldsymbol{d}$ extended with b_j on x_j.

As the embedded OR in these two cases are identical to the ones when assigning a value to $h(0, \ldots, 0)$ creates an embedded OR in h, we refer to Figure 17 for illustrations of the cases.

Case 4.2: h depends on all its inputs.

This last case requires careful analysis. We pick three variables $x_1, x_2, x_3 \in X_1$. As $|X_1| \geq 3$ we need to distinguish between when $|X_1| = 3$ and when $|X_1 > 3$. We show all three cases in Figure 18.

We begin with $|X_1| > 3$. In this case, there is an embedded OR with $X'_1 = \{x_1, x_2\}$ and $X'_2 = X_2 \cup \{x_3\}$. Define $\boldsymbol{a}' = \{a_1, a_2\}$ and $\boldsymbol{b}' = \{b_1, b_2\}$. Let $\boldsymbol{c}' = \boldsymbol{c}$ extended by $\{a_3\}$ and $\boldsymbol{d}' = \boldsymbol{d}$ extended by $\{b_3\}$. Let $\boldsymbol{e}'$ be $\boldsymbol{e}$ extended by the values from $\boldsymbol{a}$ for $X_1 \backslash \{x_1, x_2, x_3\}$. We verify that $f^{\boldsymbol{e}'}_{\{X'_1, X'_2\}}(\boldsymbol{a}', \boldsymbol{c}') = \alpha$ and $f^{\boldsymbol{e}'}_{\{X'_1, X'_2\}}(\boldsymbol{a}', \boldsymbol{d}') = f^{\boldsymbol{e}'}_{\{X'_1, X'_2\}}(\boldsymbol{b}', \boldsymbol{c}') = f^{\boldsymbol{e}'}_{\{X'_1, X'_2\}}(\boldsymbol{b}', \boldsymbol{d}') \neq \alpha$.

When $|X_1| = 3$ we get two cases depending on whether $g(\{x_1\}, \boldsymbol{c}) = \beta$ or $g(\{x_1\}, \boldsymbol{c}) = \gamma$. In the latter case, we claim that the exact same embedded OR as we used for $|X_1| > 3$ exists in f with $f^{\boldsymbol{e}'}_{\{X'_1, X'_2\}}(\boldsymbol{a}', \boldsymbol{d}') = f^{\boldsymbol{e}'}_{\{X'_1, X'_2\}}(\boldsymbol{b}', \boldsymbol{c}') = f^{\boldsymbol{e}'}_{\{X'_1, X'_2\}}(\boldsymbol{b}', \boldsymbol{d}') = \beta$.

Finally, when $|X_1| = 3$ and $g(\{x_1\}, \boldsymbol{c}) = \beta$, we claim there is an embedded OR with $X'_1 = \{x_1\}$ and $X'_2 = X_2 \cup \{x_2\}$. We let $\boldsymbol{a}' = (b_1)$, $\boldsymbol{b}' = (a_1)$, let $\boldsymbol{c}'$ be $\boldsymbol{c}$ extended with a_2, $\boldsymbol{d}'$ be $\boldsymbol{d}$ extended with b_2 and $\boldsymbol{e}'$ be $\boldsymbol{e}$ extended with b_3. We can verify $f^{\boldsymbol{e}'}_{\{X'_1, X'_2\}}(\boldsymbol{a}', \boldsymbol{c}') = \gamma$ and $f^{\boldsymbol{e}'}_{\{X'_1, X'_2\}}(\boldsymbol{a}', \boldsymbol{d}') = f^{\boldsymbol{e}'}_{\{X'_1, X'_2\}}(\boldsymbol{b}', \boldsymbol{c}') = f^{\boldsymbol{e}'}_{\{X'_1, X'_2\}}(\boldsymbol{b}', \boldsymbol{d}') = \beta$.

$\square$

$S =$	$\boldsymbol{x} = \boldsymbol{c}$	$\boldsymbol{x} = \boldsymbol{d}$
$\emptyset$	$\boldsymbol{\alpha}$	α
$\{x_1\}$	β	γ
$\{x_2\}$	β	γ
$\{x_3\}$	β	$\boldsymbol{\gamma}$
$\{x_1, x_2\}$	$\boldsymbol{\gamma}$	β
$\{x_1, x_3\}$	γ	β
$\{x_2, x_3\}$	γ	β
$\{x_1, x_2, x_3\}$	β	$\boldsymbol{\gamma}$

(a) f_1

$S =$	$\boldsymbol{x} = \boldsymbol{c}$	$\boldsymbol{x} = \boldsymbol{d}$
$\emptyset$	$\boldsymbol{\alpha}$	α
$\{x_1\}$	γ	β
$\{x_2\}$	γ	β
$\{x_3\}$	γ	$\boldsymbol{\beta}$
$\{x_1, x_2\}$	$\boldsymbol{\beta}$	γ
$\{x_1, x_3\}$	β	γ
$\{x_2, x_3\}$	β	γ
$\{x_1, x_2, x_3\}$	α	$\boldsymbol{\beta}$

(b) f_2

$S =$	$\boldsymbol{x} = \boldsymbol{c}$	$\boldsymbol{x} = \boldsymbol{d}$
$\emptyset$	α	α
$\{x_1\}$	β	γ
$\{x_2\}$	β	γ
$\{x_3\}$	$\boldsymbol{\beta}$	γ
$\{x_1, x_2\}$	γ	β
$\{x_1, x_3\}$	γ	$\boldsymbol{\beta}$
$\{x_2, x_3\}$	$\boldsymbol{\gamma}$	β
$\{x_1, x_2, x_3\}$	α	$\boldsymbol{\gamma}$
X_1	α	$\boldsymbol{\beta}$

(c) f_3

Fig. 18. The three cases of Case 4.2. In f_1, $|X_1| > 3$. In f_2, $|X_1| = 3$ and $g(\{x_1\}, \boldsymbol{c}) = \beta$, and in f_3, $|X_1| = 3$ and $g(\{x_1\}, \boldsymbol{c}) = \gamma$.

III

Secure Multi-Party Sorting and Applications

Kristján Valur Jónsson[1], Gunnar Kreitz[2], and Misbah Uddin[2]

[1] Reykjavik University
[2] KTH—Royal Institute of Technology

Abstract. Sorting is among the most fundamental and well-studied problems within computer science and a core step of many algorithms. In this article, we consider the problem of constructing a secure multi-party computing (MPC) protocol for sorting. Our protocol builds on previous work in the fields of MPC and sorting networks.

Apart from the immediate uses for sorting, our protocol can be used as a building-block in more complex algorithms. We present a weighted set intersection algorithm, where each party inputs a set of weighted elements and the output consists of the input elements with their weights summed. As a practical example, we apply our protocols in a network security setting for aggregation of security incident reports from multiple reporters, specifically to detect stealthy port scans in a distributed but privacy preserving manner. Both sorting and weighted set intersection use $\mathcal{O}(n \log^2 n)$ comparisons in $\mathcal{O}(\log^2 n)$ rounds with practical constants.

Our protocols can be built upon any secret sharing scheme supporting multiplication and addition. We have implemented and evaluated the performance of sorting on the Sharemind secure multi-party computation platform, demonstrating the real-world performance of our proposed protocols.

Keywords. Secure multi-party computation; Sorting; Aggregation; Co-operative anomaly detection

1 Introduction

Secure Multi-party Computation (MPC) has been extensively studied since its introduction by Yao [27]. Briefly, MPC allows joint but private computation of some result without the parties involved being required to reveal their individual inputs.

In this work, we study MPC sorting and generalized set intersection protocols, as well as practical applications thereof. We consider the joint evaluation by n parties of a public n-ary function f in such a way that no collusion of parties learns anything more than what they do by their own inputs and observing the output. We consider the *symmetric* case, where all participants receive the same output.

1.1 Our contribution

Our contributions presented in this paper are as follows: We first present a novel secure MPC sorting algorithm. Based on secure sorting, we construct a protocol for the weighted set intersection problem. We apply our MPC protocols to the specific problem of secure cooperative processing of intrusion detection system (IDS) incident logs. Our sorting algorithm was implemented on the Sharemind MPC platform and tested on a three machine cluster run by the Sharemind team. In this setting, we can sort 2^{14} elements in under 4 minutes. Our performance evaluation demonstrates that applying MPC to the problem of secure processing of incidence reports is feasible, especially if we can assume small clustering of targets, as suggested by the empirical findings of Katti *et al.* [17].

1.2 MPC protocols

We first present a novel secure sorting protocol, which uses established MPC primitives along with techniques from the field of sorting networks [2]. Specifically, we implement a sorting network using a secure comparison protocol [12,22] to construct comparison gates. Our core contribution in MPC sorting is a relatively straightforward combination of these two existing lines of research. However, we believe our protocols and applications thereof to be novel. The fact that multi-party protocols for sorting has been stated as an open problem [9,13] supports this claim.

Apart from being of independent interest, sorting is an important building block in algorithm design. We apply our MPC sorting protocol to develop a protocol for the weighted set intersection problem. Our protocol also solves related problems, such as the top-k problem [9]. The protocol begins by sorting all inputs ordered by keys, proceeds to aggregate the counts, then sorts the records in descending order by aggregated count. For the top-k problem, the top k records are then revealed. Our algorithm is an approach to solving the generalized set intersection problem, as discussed in the MPC context by Many [21].

1.3 Secure Processing of IDS Logs

We apply our MPC protocols to the problem of aggregating network anomaly logs in a secure and privacy preserving manner. Specifically, we use our algorithms to aggregate anomaly log records from multiple mutually distrustful parties to aid in detection of malicious activity.

Example 1 (Joint intrusion detection). Many organizations run an Intrusion Detection System (IDS) to detect malicious traffic on their networks. Logs and alerts from such systems are typically kept private, but much could be gained by sharing such information if the privacy and security concerns could be solved [25]. Another indication of the importance of sharing intrusion detection logs is the `www.dshield.org` service operated by the Internet Storm Center, where firewall logs can be submitted, albeit potentially revealing sensitive internal information. By using protocols for private aggregation, cooperative processing of logs can be done in a fully automated way, and in close to real-time, as discussed by Burkhart and Dimitropoulos [9]. This can for example be used to detect an attacker running a slow scan of multiple targets in the hope of avoiding detection, or to proactively block traffic from a source flagged as being malicious by others' IDS's.

We have implemented MPC sorting protocols on the Sharemind [6] platform for secure multi-party computation and present the results of our performance evaluations on the project's test cluster. The run-time of our weighted set intersection protocol is dominated by sorting (twice), so its performance is closely connected to that of sorting. However, the protocols are not specific to the Sharemind platform. Rather, our protocols are described as a sequence of computations on secret shared data, taking input and producing output in secret shared form. This allows our protocols to be easily re-used as parts of other protocols and to be implemented on other common MPC platforms.

Our performance evaluations, presented in this paper, demonstrate the feasibility of using MPC protocols for privately processing incident logs produced by IDS systems. We reserve optimization and scaling issues with regard to our protocols for future work.

1.4 Other Applications of Secure Sorting

MPC sorting has various other direct applications. For instance, a multi-party sorting algorithm could be used as a part of a voting system by sorting all votes, which would remove the possibility of re-identifying voters. However, secure voting is a complex problem, and there is a rich literature on the subject. Other mechanisms, such as mix-nets can also be used to more efficiently prevent de-anonymization of the cast votes. By sorting and selectively opening key-value pairs of (bid, bidder)-pairs, different auction mechanisms can also be implemented by sorting. For instance, second price (Vickrey) auctions can be computed by sorting bids in descending order and then opening the bidder part of the first pair and the bid part of the second pair.

The weighted set intersection protocol we develop has many different application apart from Example 1, of which we note a few. Firstly, it can easily be modified to solve the top-k problem where only the top k records are revealed. It also solve set intersection, which has many real-life applications. One such example is given in Example 2.

Example 2 (Disease detection). A blood bank and a diagnostic center could run an intersection algorithm to determine if any person who has donated blood has later been diagnosed with a disease which may spread through blood donations. While donors are typically screened before blood donations are made, running a protocol could assist in detecting if any errors or false negatives have occurred. Health care providers often operate under very strict privacy restrictions which may prevent computing the intersection without the strong privacy guarantees provided by a secure multi-party protocol.

1.5 Related Work

Protocols to securely evaluate any function are given by Ben-Or, Goldwasser, and Wigderson [5], and Chaum, Crépeau, and Damgård [11]. Both these results present protocols for computing addition and multiplication (XOR and AND) on values in (verifiable) secret shared form, and with results remaining secret shared. As these primitives are complete, any function can then be evaluated gate by gate.

Adding to these primitives, Damgård *et al.* [12] presented a generic comparison protocol which can be used with most common secret sharing mechanisms. Later, a more efficient protocol for comparison was proposed by Nishide and Ohta [22]. Bogdanov *et al.* [7] gave efficient protocols specialized for the case when exactly 3 parties participate in the computation.

A line of research has focused on developing efficient MPC protocols for specific functions. Among these are more specific applications, such as auctions [8], and comparing gene sequences [16], but also more generic primitives such as set operations [14,18], top-k queries [9], and weighted set intersection [21]. Finding efficient algorithms for private sorting has been stated as an open problem by Du and Atallah [13], and more recently by Burkhart and Dimitropoulos [9].

A number of frameworks and specialized programming languages to implement and run secure multi-party computation protocols have been created. These include FairplayMP [4], Sharemind [6], SEPIA [10], and VIFF. FairplayMP builds on the idea of "garbled circuits" [28,3], Sharemind uses additive sharing over a ring, and the latter two systems build on Shamir's secret sharing [23]. Despite these three different approaches to implementing a generic MPC framework, they all support a similar set of primitives, including addition, multiplication, comparisons and equality testing. However, the performance properties of FairplayMP are different from the others, and we will focus on the latter three. Programming on these platforms either uses a specialized language, or a standard programming language and library calls, depending on the platform.

One of our proposed applications is a protocol for cooperative, but private, processing of security-relevant information, shared between mutually distrustful organizations. Secure sharing of server logs and incidents reports is discussed by Lincoln *et al.* [20], Slagell *et al.* [25] and Xu *et al.* [26]. The common thread is that sharing of incident logs can aid in early detection of anomalies, but that naive merging of logs is not a feasible solution, as this may reveal sensitive internal and customer information. The authors cited propose a variety of log sanitization techniques to protect potentially sensitive information. In contrast, we propose to use MPC protocols for privacy preserving cooperative log processing. Katti *et al.* [17] perform empirical analysis on a large dataset obtained from 1700 IDS systems. They observe that coordinated attacks on multiple targets are a large fraction of the total attacks observed. Further, such attacks are seen to target small clusters of victims with relatively constant size and membership. The clustering phenomenon is most likely due to the combination of software run and services offered on the platforms under attack. The observations of

Katti *et al.* support the feasibility of cooperative anomaly detection using MPC protocols, such as is the focus of our work.

Building on their SEPIA platform, Burkhart and Dimitropoulos [9] have proposed an algorithm for aggregation and top-k queries. In this paper, we present an alternate algorithm for top-k queries based on sorting. We remark that their solution computes an approximately correct value, while our proposed solution computes exact results.

Also building on the SEPIA platform, Many [21] in his Master's thesis studied a problem called weighted set intersection. In this problem, each participant gives as input a list of (value, weight)-pairs, and only keys reported by some number of peers and which have weights with sum greater than some threshold are output. For this problem, we propose an alternate solution based on sorting.

We apply techniques from sorting networks in our primary building block, the MPC sorting protocol. A sorting network is a circuit which uses a fixed number of comparison gates. A comparison gate is a gate with two inputs a, b and two outputs, which output $\min(a, b)$ and $\max(a, b)$. Another perspective is that a sorting network is a sorting algorithm with the property that which elements it compares is independent of the input.

Two performance metrics are of importance in the sorting network literature, as well as in our application: the number of comparison gates, and the depth of the circuit. Batcher [2] presented two famous sorting networks, odd-even merge sort and bitonic sort. Both algorithms have depth $\mathcal{O}(\log^2 n)$ using $\mathcal{O}(n \log^2 n)$ gates and are efficient in practice. The Shell sort [24] algorithm can also be implemented as a sorting network with the same performance. Ajtai, Komlós, and Szemerédi [1] constructed the AKS sorting network which achieves the theoretically optimal $\mathcal{O}(\log n)$ depth with $\mathcal{O}(n \log n)$ gates, but the constant hidden in the ordo notation makes the algorithm inefficient for practical input sizes. Leighton and Plaxton [19] have proposed a sorting network with optimal asymptotic performance and practical constants, but which do not sort correctly for a small fraction of inputs.

2 Preliminaries and Notation

We denote a sequence of elements by a bold letter, like $\boldsymbol{a}$, and use the same letter with an index to denote the elements in the sequence, so a_i is the i^{th} element of the sequence $\boldsymbol{a}$. When we treat sequences of pairs, we explicitly associate two new letters with the first and second item of the pair. Thus, for a vector $\boldsymbol{b}$ consisting of pairs (x_i, y_i) the element $b_j = (x_j, y_j)$.

2.1 Secure Multi-Party Computation

The most common paradigm for MPC is that of computing on shared secrets [5,11]. In this paradigm, parties use secret sharing to share their private inputs among all participants in the computation. They then execute protocols which operate on secret shared inputs and return the output in secret shared form.

Finally, the shares of the output are combined to recover the output of the function.

Our protocols are based on a small number of primitives, which we assume to be securely implemented. These operations are addition, multiplication, comparison and equality testing, operating on values under secret sharing. The values operated on are in a finite ring or field, which is known to all parties. We denote the ring or field by $\mathbb{Z}_k$. In Table 1, we list the primitives and the notation we use. All arithmetic operations are in $\mathbb{Z}_k$.

Notation	Operation
$[x]$	The value x shared between the parties
$[\boldsymbol{a}]$	The sequence $\boldsymbol{a}$ shared element-wise between the parties
$[x] + [y]$	Computation of the value $x + y$ shared by the parties
$c \cdot [x]$	Computation of x multiplied by a public constant
$[x] \cdot [y]$	Computation of the product of two secret shared values
$[x < y]$	The value 1 if x is strictly smaller than y, 0 otherwise
$[x = y]$	The value 1 if x is equal to y, 0 otherwise

Table 1. List of MPC primitives used

Several of our algorithms operate on sequences. When sharing a sequence, it is shared element-wise. Thus, the notation $[\boldsymbol{a}]$ for $\boldsymbol{a}$ of length m means that the parties have shares $[a_1], [a_2], \ldots, [a_m]$. We remark that sharing a sequence reveals the length of the sequence to all parties. Similarly, when sharing pairs we share them element-wise. Thus, for a sequence $\boldsymbol{b}$ of length m consisting of pairs (x_i, y_i), $[\boldsymbol{b}]$ means that parties have shares $[x_1], [y_1], [x_2], [y_2], \ldots, [x_m], [y_m]$.

The computational and communication cost of the different primitives varies significantly. Addition of two shares, as well as multiplication by a constant, can be performed locally by all participants in the protocol. Computing the less-than function or equality is done by computing a large number of multiplications, the exact number depending on the protocol used and on the bit length of the elements. For instance, the protocol by Nishide and Ohta [22] uses 15 rounds and $279\ell + 5$ multiplications to compare two elements where ℓ is the number of bits needed to represent an element in $\mathbb{Z}_k$, i.e. $\lceil \log_2 k \rceil$. All operations in Table 1 use a constant number of rounds. For fixed k, the communication is also $\mathcal{O}(1)$.

As in other areas of computer science, MPC protocols benefit from parallelization. Therefore, when designing and implementing a protocol, it is important to keep not only the number of operations, but also the number of rounds as low as possible. In Section 5, we demonstrate the importance of parallelization by comparing the run time of a parallelized and non-parallelized version of bubble sort.

In the context of MPC, different notions of what a party is has been discussed. Traditionally, the model has been that all parties contributing input also participate in the secure computation. However, as performance of MPC protocols does not scale well to a large number n of participants, a different model has been proposed [6,10]. In this model, a large number of *peers* collect data and send it in secret shared form to a relatively small number of *privacy peers*. The privacy peers then run the protocol to perform the secure computation on the inputs. Most often when designing protocols, the distinction between the two models is not important. However, as we discuss in Section 3, for one of our sorting protocols, it does make a difference.

The exact security guarantees achieved, as well as the constraints on the execution environment depends on the MPC framework on which our protocol is implemented. The frameworks based on Shamir's secret sharing are secure against a *passive* adversary who can corrupt up to $\lfloor (n-1)/2 \rfloor$ nodes. These protocols require each pair of computing nodes to be connected with *private channels*. The security guarantee still holds even if the adversary has unlimited computational power (*information-theoretic security*). The Sharemind framework is implemented for $n = 3$ parties and provides security against a passive adversary corrupting any single node. By implementing the protocols using verifiable secret sharing, security against up to $\lfloor (n-1)/3 \rfloor$ *active* adversaries can be achieved. Of the sharing-based frameworks, only VIFF has implemented verifiable secret sharing.

Informally, the security guarantees in the passive case say that no collusion of parties (for collusions smaller than the security threshold) learn more than what they do from their inputs and the output of the function. We refer to [15] for the formal definitions.

2.2 Sorting Networks

A sorting network is a circuit with m inputs and m outputs[3]. The outputs take the same values as the inputs, but in sorted order. The gates used to construct the circuit are so called *comparison gates*, which compare and sort two elements. We may view a sorting network as a sorting algorithm with the property that the comparisons made are independent from the values to be sorted. Among the well-known traditional sorting algorithms, bubble sort and Shell sort can both be implemented in such a way that they have this property.

The viewpoint of a sorting network as a sorting algorithm with a fixed sets of comparisons illuminates why such constructions can function as MPC sorting algorithms. This means that given a design for a comparison gate, which takes two inputs in secret shared form and sorts them (keeping them in secret shared form), a sorting network can be implemented as an MPC computation. Furthermore, the performance metrics of a sorting network are the same as those we are interested in optimizing in MPC; the circuit depth corresponds to the number of rounds and the number of comparison gates used to the number of comparisons.

A comparison gate is a gate with two inputs a, b and two outputs, h, l such that the output h takes value $\max(a, b)$ and the output l takes value $\min(a, b)$.

[3] More formally, it is a family of circuits for all sizes m, but we will ignore this distinction.

A comparison gate can be implemented as a multi-party computation by using the following construction:

$$[h] = [a < b] \cdot [b] + (1 - [a < b]) \cdot [a]$$
$$[l] = [a < b] \cdot [a] + (1 - [a < b]) \cdot [b]$$

We define the function Compare-exchange(a_i, a_j) and assume sequences are sorted in-place. This function compares the two elements a_i, a_j and swaps them if needed such that after its execution, $a_i \leq a_j$. The Compare-exchange operation is implemented using the construction given above. We present the algorithm later in Algorithm 3.

Sorting networks are comparison based, so the general $\Omega(m \log m)$ lower bound on the number of comparisons applies. The AKS sorting network [1] matches this bound, but with such large constants that it is inefficient for practical input sizes. There are several sorting networks [2,24] with slightly worse asymptotic performance, $\mathcal{O}(m \log^2 m)$ comparisons and depth $\mathcal{O}(\log^2 m)$, but with good constants. Of these, we will focus on the odd-even merge sort algorithm by Batcher [2]. There is a also fast sorting network by Leighton and Plaxton [19] with $\mathcal{O}(m \log m)$ comparisons, but which does not correctly sort the output for a very small fraction of the possible inputs. In the context of MPC, we feel that correctness outweighs performance. As we show in Section 5, an MPC implementation of odd-even merge sort achieves good performance even on reasonably large data sets.

As the name implies, odd-even merge sort is based on the well-known merge sort algorithm. The overall structure is the same, splitting the input in two halves, recursively sorting each half, and then merging. The difference compared to the standard merge sort algorithm lies in the merge step, which we describe in Algorithm 1. For completeness, we give a full description of the whole algorithm in Algorithm 2. With some abuse of notation, we write Function$(a_1, a_2, \ldots, a_m)$ to call Function with the sequence $\boldsymbol{a} = a_1, a_2, \ldots, a_m$ when Function takes a single sequence as argument. We describe the algorithms as operating on a sequence, the length of which is a power of 2, but it is easy to modify to arbitrary input lengths by omitting some comparisons. We conclude this section by restating the correctness and performance analysis from [2] of the odd-even merge sort algorithm as a theorem.

Theorem 1 ([2]). *The* Odd-even merge *algorithm uses* $\mathcal{O}(m \log m)$ *comparisons with depth* $\mathcal{O}(\log m)$*.* Odd-even merge sort *correctly sorts using* $\mathcal{O}(m \log^2 m)$ *comparisons with depth* $\mathcal{O}(\log^2 m)$*.*

3 Secure Multi-Party Sorting

We construct a secure sorting network on secret shares by using a MPC protocol for Compare-exchange. This allows us to leverage the long line of research in

Algorithm 1 Odd-even merge [2]

```
Input: Sequence $\boldsymbol{a}$ whose two halves $a_1, a_2, \ldots a_{m/2}$ and $a_{m/2+1}, a_{m/2+2}, \ldots, a_m$ are sorted. Length $m$ is a power of 2.
Output: Sequence $\boldsymbol{a}$ is modified in-place to be sorted.
if $m > 2$ then
    Odd-even merge($a_1, a_3, a_5, \ldots, a_{m-1}$)
    Odd-even merge($a_2, a_4, a_6, \ldots, a_m$)
    for $i \in \{2, 4, \ldots, m-2\}$ do
        Compare-exchange($a_i$, $a_{i+1}$)
    end for
else
    Compare-exchange($a_1$, $a_2$)
end if
```

Algorithm 2 Odd-even merge sort [2]

```
Input: Sequence $\boldsymbol{a}$ of length $m$ (a power of 2).
Output: Sequence $\boldsymbol{a}$ is modified in-place to be sorted.
if $m > 1$ then
    Odd-even merge sort($a_1, a_2, \ldots, a_{m/2}$)
    Odd-even merge sort($a_{m/2+1}, a_{m/2+2}, \ldots, a_m$)
    Odd-even merge($\boldsymbol{a}$)
end if
```

sorting networks to construct efficient MPC sorting protocols. However, there are a number of details which remain in constructing an MPC sorting algorithm related to the input to the protocol which we proceed to discuss.

3.1 Problem Definition

When we discuss MPC sorting, there are actually two slightly different problems which we tackle. The first is sorting as a stand-alone functionality, or as the first step of an algorithm. In this usage, we may need to hide the number of items which are contributed by each party. The second is using sorting as a step within an algorithm, where the number of elements to be sorted is known. We give a definition to capture this distinction.

Definition 1 (Multi-party sorting, composable, public input lengths). *A multi-party sorting protocol is run between n parties. Each party P_i gives as input a sequence $\boldsymbol{a}^i$ of length m_i. After the protocol has executed, the parties learn the sorted sequence $\boldsymbol{a}$, whose elements are the concatenations of the sequences $\boldsymbol{a}^i$. We say that a sorting protocol is* composable *if the parties learn the sorted output $\boldsymbol{a}$ in secret shared form and the total length $|\boldsymbol{a}| = \sum_{i=1}^{n} m_i$ openly. If the parties learn the input lengths contributed by other parties, we say that the protocol has* public input lengths*, otherwise we say it has* private input lengths.

All of our protocols are given as standard MPC operations on secret shares, and are thus composable. The intuition for the notion of public input lengths comes from the fact that it is difficult to hide the size of inputs in the context of MPC protocols. For sorting, we give a general, but somewhat costly, transformation which transforms a protocol with public input lengths into a protocol with private input lengths. We stress that even with public input lengths, which party contributed what element of the sorted output remains secret.

3.2 Public and Private Input Lengths

Our transformation from public to private input lengths begins by the parties computing the length of the output. This is easily done by each party secret sharing the length of their input. The lengths are summed, and the result is opened and revealed to all parties. Let the sum be m. Let 0 be the smallest possible input. Each party then locally appends 0's to their input until their input is of length m. They then participate in sorting using this new, longer input. The cost is that the sorting algorithm must be run on a sequence of length mn (where n is the number of parties) instead of length m.

Here, we would like to tie back into the discussion on the roles of different parties in the MPC context. This transformation is slightly at odds with the privacy-peer view of MPC, as it requires one round of interaction between the data reporting peers and the privacy-peers executing the MPC protocol, as the peers must first contribute their number of shares, wait to receive the sum of these numbers, and then contribute their actual input (appropriately padded). However, we believe this limited interaction is acceptable in most applications of MPC in a privacy-peer setting as the interaction is quick, very limited and not computationally demanding for the data reporting peers.

A similar issue arises with regard to the domain in which the elements lie. For our MPC protocol, we require the elements to be in $\mathbb{Z}_k$ for some public k. But how large is k? In most applications, an upper bound on k is known *a priori*, for instance when operating on IP addresses which are of fixed length. If no such bound is known from the applications, the parties can precede the sorting protocol with a protocol computing the maximum element in the input to some suitable degree of precision (e.g., the number of bits required to store it) and then instantiate the MPC framework with a ring or field of appropriate size. In the remainder of this paper, we assume that the parties know a suitable size *a priori* to present the main ideas more clearly.

3.3 Multi-party Sorting

We are now ready to present our protocol for MPC sorting. We begin by describing a comparison gate, COMPARE-EXCHANGE. We present the algorithm as operating on key-value pairs in Algorithm 3. With some abuse of notation, we call this function both when sorting key-value pairs and when sorting singleton elements. In the algorithm, we use $[s]$ and $[t]$ as temporary variables for the values that will go into outputs $[x_1]$ and $[y_1]$.

Algorithm 3 COMPARE-EXCHANGE

```
**Input:** Two key-value pairs $([x_1], [y_1])$, $([x_2], [y_2])$ in secret shared form
**Output:** The pairs are swapped in-place such that the pair with the smallest key is in the first position.
Compute $[c] \leftarrow [x_1 < x_2]$
Let $[s] \leftarrow [c] \cdot [x_1] + (1 - [c]) \cdot [x_2]$
Let $[t] \leftarrow [c] \cdot [y_1] + (1 - [c]) \cdot [y_2]$
Let $[x_2] \leftarrow (1 - [c]) \cdot [x_1] + [c] \cdot [x_2]$
Let $[y_2] \leftarrow (1 - [c]) \cdot [y_1] + [c] \cdot [y_2]$
Let $[x_1] \leftarrow [s]$
Let $[y_1] \leftarrow [t]$
```

We include a full description of the protocol as Algorithm 4. The construction used for private input lengths is general and can be used for any MPC sorting protocol.

We remark that the users could sort their own contributed inputs locally. Some sorting networks, in particular the odd-even merge sort network that we presented as Algorithm 2 could be modified to make use of this to increase the performance. However, we omit the details of such modifications.

Algorithm 4 MULTI-PARTY SORTING

```
**Input:** Each party $P_i$ inputs sequence $\boldsymbol{a}^i$ of length $m_i$
**Output:** Sorted sequence $\boldsymbol{a}$ in secret shared form
**if** Private input lengths **then**
    Each party $P_i$ shares $[m_i]$
    Compute $[m] = \sum_{i=1}^{n} [m_i]$
    Open $[m]$ to all parties
    Each party $P_i$ forms $\boldsymbol{b}^i$ by padding $\boldsymbol{a}^i$ with $m - m_i$ 0 elements
**else**
    Each party $P_i$ sets $\boldsymbol{b}^i \leftarrow \boldsymbol{a}^i$
**end if**
Each party $P_i$ shares $\boldsymbol{b}^i$ element-wise
Let $[\boldsymbol{b}]$ be the concatenation of all $[\boldsymbol{b}^i]$
Sort $[\boldsymbol{b}]$ in-place using a sorting network
**if** Private input lengths **then**
    **return** the last $m$ elements of $\boldsymbol{b}$
**else**
    **return** $\boldsymbol{b}$
**end if**
```

Theorem 2. *Algorithm 4 is correct and secure.*

Proof (Sketch). Correctness of the algorithm follows directly from the correctness of the sorting network implemented.

The security of the protocol relies on the security of the implementation of the underlying MPC primitives for all MPC operations. In the case with private input lengths, each party performs the same sequence of MPC operations independent of their inputs. Without private input sizes, the only difference between parties in the protocol is how many elements are shared by each party. The only value opened to the parties is the size of the output, which is part of the output in Definition 1. □

4 Weighted Set Intersection and Aggregation

We now return to our motivating example of a joint IDS, Example 1. We claim that a useful tool for IDS cooperation is one where each party contributes a list of suspected attackers, and a weight indicating their confidence that it is the source of an attack. The weights may naturally come from simple counting, such as the number of TCP SYN packets seen from a host to more complex combinations of multiple criteria. When multiple IDS's report the same suspected attacker, their weights need to be aggregated. We constrain our discussion to the simplest aggregation: summing the weights. Depending on the exact application, one may then wish to reveal to all parties the suspects over some fixed threshold, the top-k suspects, or some other function of the list. We think of an entry with weight 0 as empty and do not include such entries in the output.

As motivation for selecting this problem, Many [21] contains an in-depth discussion on primitives which may be of use in joint intrusion detection. They develop a protocol for generalized set intersection, a problem which is closely related to the problem we solve. They give a good motivation for its applicability in the IDS context. Further motivation is given by the observation by Katti *et al.* [17] that a large fraction of attackers attack targets within a few minutes of each other, indicating that if entities could collaborate on intrusion detection and prevention in near real time, attacks could be proactively blocked.

4.1 Problem Definition

We formally define our problem and refer to it as weighted set intersection. We remark that this name was also used by Many [21] for their problem, which is closely related but not exactly the same. In light of the application, we will refer to the weighted set intersection or the top-k problem as *aggregation*.

Definition 2 (Weighted Set Intersection, Top-k). *In the Weighted Set Intersection problem, each party P_i gives as input a sequence $\boldsymbol{a}^i$ of length m_i consisting of value-weight pairs, $a^i_j = (v^i_j, w^i_j)$, with positive weights. The output is a sequence of value-weight pairs such that element v_j is in the output if it was in the input of any party. Elements in the output have as weight the sum of weights it occurred with in the input of all parties. If the output is truncated to only include the top k values, sorted by their aggregate weights, we say that it solves the top-k problem.*

In Section 3, we defined and discussed the notion of public or private input lengths. A similar concern exists in our current setting. However, this time, the padding idea we used in sorting does not work. The key difference between sorting and aggregation is that with aggregation, it is difficult to compute the length of the output without performing the actual aggregation. By applying the same transformation as we did for sorting, we would get a protocol hiding the number of items submitted by each party, but where the difference between the length of the inputs and the output reveals how many key-value pairs were reported by more than one party. Thus, we propose a protocol for aggregation with public input lengths.

4.2 The Core Aggregation Step

In our algorithm, we sort the complete list of item-weight pairs, ordered on item. Then an aggregation step is run, which for all items aggregates their weights. After the aggregation step, we want a single entry in the list for each distinct item, its weight being the sum of its weights in the original input. This means that some item-weight pairs need to be removed; "removal" of an item-weight pair is performed by setting its weight to 0 which ensures its exclusion from the output later in the algorithm. We begin our description by presenting in Algorithm 5 an MPC algorithm, AGGREGATE-IF-EQUAL, for comparing two key-value pairs and merging them if they have the same key, analogous in function to COMPARE-EXCHANGE.

Algorithm 5 AGGREGATE-IF-EQUAL

```
Input: Two key-value pairs ($[x_1], [y_1]$), ($[x_2], [y_2]$) in secret shared form
Output: If the keys are equal, the values are merged in-place
Compute $[c] \leftarrow [x_1 = x_2]$
Let $[y_1] \leftarrow [y_1] + [c] \cdot [y_2]$
Let $[y_2] \leftarrow (1 - [c]) \cdot [y_2]$
```

Returning to the original question: given a sorted list, how to aggregate the weights for an item? A naive solution would be to use $\mathcal{O}(m^2)$ calls to AGGREGATE-IF-EQUAL, comparing each pair of items and summing weights appropriately. However, we can do better. We present an algorithm which aggregates m items using $\mathcal{O}(m \log m)$ equality tests with a construction similar to Batcher's ODD-EVEN MERGE algorithm. We call this algorithm ODD-EVEN AGGREGATION and give the pseudo-code in Algorithm 6 and proof of correctness in Theorem 3. As with sorting, we describe the algorithm for an input lengths which is a power of 2, and the algorithm can easily be adapted to arbitrary input lengths.

Theorem 3. *Algorithm 6 correctly aggregates its input using $\mathcal{O}(m \log m)$ equality tests in $\mathcal{O}(\log m)$ rounds.*

Algorithm 6 Odd-even aggregation

```
Input: Sequence $\boldsymbol{a}$ of key-value pairs $(x_i, y_i)$ which are sorted by the key values.
    The length $m$ of $\boldsymbol{a}$ is a power of 2.
Output: Sequence $\boldsymbol{a}$ is modified in-place to contain one key-value pair with non-zero
    weight for each unique key. "Removed" entries have weight set to 0.
if $m > 2$ then
    Odd-even aggregation$(a_1, a_3, a_5, \ldots, a_{m-1})$
    Odd-even aggregation$(a_2, a_4, a_6, \ldots, a_m)$
    for $i \in \{1, 2, 3, \ldots, m-1\}$ do
        Aggregate-if-equal$(a_i,\ a_{i+1})$
    end for
else
    Aggregate-if-equal$(a_1,\ a_2)$
end if
```

Proof (Sketch). We observe that on inputs of the same length, Odd-even aggregation runs Aggregate-if-equal twice as many times as Odd-even merge runs Compare-exchange. From Theorem 1, the performance follows.

The proof of correctness is by induction on the input length. We claim that the aggregated entry for a key x that occurs in the input with non-zero weight will be stored at the first position where x occurred in the input, i.e. the smallest i such that $x_i = x$. The base case of input length 2 is easily verified.

We observe that since Aggregate-if-equal compares the keys of elements, calling it on items with different keys does not cause problems. As it overwrites the weight with 0 when merging, calling it on a position which has already been aggregated is also harmless. Thus, our main concern is proving that all entries with the same key will indeed be aggregated into the first entry with that key.

The two recursive calls (on the odd and even sub-sequences) are on sorted sequences, so by induction the two sub-sequences are correctly aggregated. For a key x occurring in the input, let i be the least even i such that $x_i = x$, and let j be the least odd j such that $x_j = x$ in the input. By induction, the weights for x are aggregated into positions i and j. As the input was sorted by key, we have $|i - j| = 1$, and thus the two entries are merged by the for-loop calling Aggregate-if-equal. As this holds for any x, it concludes the proof. □

4.3 Algorithm for Weighted Set Intersection

Given the algorithm for the aggregation step on a sorted list, the key building blocks for the weighted set intersection algorithm are in place. To handle the "removed" elements in the output after the aggregation step, we sort the aggregated list and count the number of non-zero items in the list, which is the number of elements to keep as output. We present the full algorithm in Algorithm 7.

Theorem 4. *Algorithm 7 correctly and privately (with public input lengths) solves the weighted set intersection problem using $\mathcal{O}(m \log^2 m)$ comparisons and $\mathcal{O}(m \log m)$ equality tests in $\mathcal{O}(\log^2 m)$ rounds.*

Algorithm 7 MPC Weighted Set Intersection

```
Input: Party $P_i$ contributes a list $\boldsymbol{a}^i$ of item-weight pairs $(x_j, y_j)$ of length $m_i$.
Output: Sequence of item-weight pairs, sorted by value
Each party $P_i$ shares its input element-wise, $[\boldsymbol{a}^i] = [x_1^i], [y_1^i], [x_2^i], [y_2^i], \dots [x_{m_i}^i], [y_{m_i}^i]$
Let $\boldsymbol{a}$ be the concatenation of the sequences $\boldsymbol{a}^i$
Odd-even merge sort($\boldsymbol{a}$), sorting on $x_j$
Odd-even aggregation($\boldsymbol{a}$)
Odd-even merge sort($\boldsymbol{a}$), sorting on $y_j$
$[m] \leftarrow 0$
for $i = 1$ to $|\boldsymbol{a}|$ do
    $[m] = [m] + (1 - [x_i = 0])$
end for
Open $[m]$ to all parties
return The $m$ last elements of $\boldsymbol{a}$
```

Proof (Sketch). Correctness of the algorithm follows from Theorem 1 and Theorem 3 (correctness of sorting and aggregation).

The security of the protocol relies on the security of the implementation of the underlying MPC primitives for all MPC operations. The number of elements shared by each party reveals the length of their input. After contributing their input, all parties perform the same operations. The only value opened to the parties is the size of the output, which is part of the output. □

5 Performance Evaluation

To demonstrate the practical applicability of our protocols, we have made a proof-of-concept implementation of our proposed sorting algorithm on the Sharemind MPC platform. The implementation is not highly optimized, but our results show that it can sort 2^{14} 32-bit values in just above 3.5 minutes. We believe that this speed could be further improved by future work either on the implementation of the sorting algorithm or on the Sharemind platform itself. As the run time of our aggregation algorithm is dominated by sorting, the performance of sorting is highly indicative of the performance of aggregation.

For comparison purposes, we have also implemented sorting based on bubble sort and evaluated its performance on the same cluster. In our bubble sort implementation, we implemented both a vectorized version and a serialized version, demonstrating the large performance gains from running operations in parallel.

All of our experiments were executed on a cluster operated by the Sharemind team. As the platform only supports 3-party computations, all tests are run using 3 computers running version 2.0 of the Sharemind framework. The computers were equipped with dual Intel X5670 CPUs 2.93 GHz CPUs and 48 GB RAM, and were interconnected by gigabit links.

The Sharemind framework has its own programming language, SecreC, that is used to specify the algorithms run on the platform. Our implementation of all measured algorithms are fully unrolled, meaning that the SecreC code contains the full program sequence without iteration or recursive calls.

In Figure 1, we show how the total execution time of the three algorithms varies as a function of the total number of items sorted, n. For technical reasons, we were not able to run experiments sorting more than 2^{14} elements.

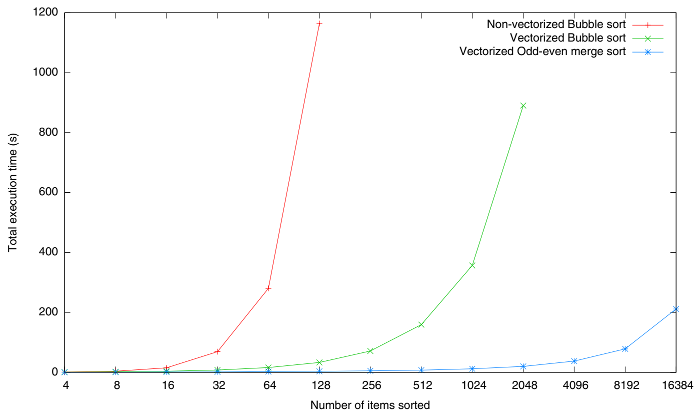


Fig. 1. Execution time (wall time) of the three compared sorting implementations. Number of elements on x-axis on log-scale.

As can be seen from Figure 1, our baseline non-vectorized bubble sort algorithm can sort 2^6 elements in a few minutes. By performing operations in parallel, bubble sort's performance increases considerably and it sorts 2^{10} elements within minutes, demonstrating the importance of parallelism in MPC computations. Not surprisingly, by using the better algorithm, we further increase practical input sizes, and with our implementation of odd-even merge sort one can sort 2^{14} elements in the same amount of time. This means that joint IDS's generating a few thousand alerts per minute can be practically implemented with near real-time performance.

6 Conclusion

We have given a description of a MPC sorting algorithm and an aggregation algorithm based on sorting. Implementing and evaluating the algorithms on an MPC programming platform, we have shown that the performance of MPC sorting is practical for reasonably large data sets.

In the context of intrusion detection, there is much to be gained by collaborating. However, the sensitive nature of alerts from an IDS makes it difficult to do so. MPC protocols offer very strong security guarantees, and as we have shown in our evaluation, are approaching practical performance for intrusion detection applications.

By using our construction, any sorting network can be implemented as an MPC program. However, it may be possible to construct MPC sorting algorithms which do not correspond to a sorting network. We leave as an open problem the question of whether there is an MPC sorting algorithm using $o(n \log^2 n)$ comparisons, which is also fast for practical input sizes.

Acknowledgments

We would like to thank the Sharemind team for their valuable feedback on our work, and for the help with the experiments.

References

1. Miklós Ajtai, János Komlós, and Endre Szemerédi. An O(n log n) sorting network. In *STOC*, pages 1–9. ACM, 1983.
2. Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.
3. Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *STOC*, pages 503–513. ACM, 1990.
4. Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 257–266. ACM, 2008.
5. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10. ACM, 1988.
6. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.
7. Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. Improved protocols for the Sharemind virtual machine. Technical Report T-4-10, Cybernetica, 2010.
8. Peter Bogetoft, Ivan Damgård, Thomas P. Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In Giovanni Di Crescenzo and Aviel D. Rubin, editors, *Financial Cryptography*, volume 4107 of *Lecture Notes in Computer Science*, pages 142–147. Springer, 2006.
9. Martin Burkhart and Xenofontas Dimitropoulos. Fast privacy-preserving top-k queries using secret sharing. In *19th International Conference on Computer Communications and Networks (ICCCN)*, Zurich, Switzerland, August 2010.
10. Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *19th USENIX Security Symposium*, Washington, DC, USA, August 2010.
11. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *STOC*, pages 11–19. ACM, 1988.

12. Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.
13. Wenliang Du and Mikhail J. Atallah. Secure multi-party computation problems and their applications: A review and open problems. In Victor Raskin, Steven J. Greenwald, Brenda Timmerman, and Darrell M. Kienzle, editors, *NSPW*, pages 13–22. ACM, 2001.
14. Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT*, volume 3027 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2004.
15. Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications.* Cambridge University Press, New York, NY, USA, 2004.
16. Somesh Jha, Louis Kruger, and Vitaly Shmatikov. Towards practical privacy for genomic computation. In *IEEE Symposium on Security and Privacy*, pages 216–230. IEEE Computer Society, 2008.
17. Sachin Katti, Balachander Krishnamurthy, and Dina Katabi. Collaborating against common enemies. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, IMC '05, pages 34–34, Berkeley, CA, USA, 2005. USENIX Association.
18. Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 241–257. Springer, 2005.
19. Frank Thomson Leighton and C. Greg Plaxton. Hypercubic sorting networks. *SIAM J. Comput.*, 27(1):1–47, 1998.
20. Patrick Lincoln, Phillip Porras, and Vitally Shmatikov. Privacy-preserving sharing and correction of security alerts. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 17–17, Berkeley, CA, USA, 2004. USENIX Association.
21. Dilip Many. Privacy-preserving collaboration in network security. Master's thesis, ETH Zürich, 2009.
22. Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *Public Key Cryptography*, volume 4450 of *Lecture Notes in Computer Science*, pages 343–360. Springer, 2007.
23. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
24. Donald L. Shell. A high-speed sorting procedure. *Commun. ACM*, 2(7):30–32, 1959.
25. Adam J. Slagell and William Yurcik. Sharing computer network logs for security and privacy: A motivation for new methodologies of anonymization. *CoRR*, cs.CR/0409005, 2004.
26. Dingbang Xu and Peng Ning. Privacy-preserving alert correlation: a concept hierarchy based approach. In *21st Annual Computer Security Applications Conference*, 2005.
27. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE, 1982.
28. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167. IEEE, 1986.

IV

Practical Private Information Aggregation in Large Networks

Gunnar Kreitz, Mads Dam, and Douglas Wikström

KTH—Royal Institute of Technology
Stockholm
Sweden

Abstract. Emerging approaches to network monitoring involve large numbers of agents collaborating to produce performance or security related statistics on huge, partial mesh networks. The aggregation process often involves security or business-critical information which network providers are generally unwilling to share without strong privacy protection. We present efficient and scalable protocols for privately computing a large range of aggregation functions based on addition, disjunction, and max/min. For addition, we give a protocol that is information-theoretically secure against a passive adversary, and which requires only one additional round compared to non-private protocols for computing sums. For disjunctions, we present both a computationally secure, and an information-theoretically secure solution. The latter uses a general composition approach which executes the sum protocol together with a standard multi-party protocol for a complete subgraph of "trusted servers". This can be used, for instance, when a large network can be partitioned into a smaller number of provider domains.

Keywords. Multi-party computation; Private aggregation; Partial mesh network

1 Introduction

With the continuous increase of network complexity and attacker sophistication, the subject of network and security monitoring becomes increasingly important. Traditionally, organizations have performed network and security monitoring based only on data they can collect themselves. One of the reasons for this is a reluctance to share traffic data and security logs between organizations, as such data is sensitive.

There is much to be gained from collaboration in security monitoring. Attacks range from being targeted at specific individuals or organizations, to global scale attacks such as botnets. Naturally, the response measures depend on the type of attack. The same situation applies to network monitoring, where the complexity of networks, and large amount of applications can make it difficult to distinguish between local and global disruptions with access only to local data.

A natural path towards a solution is to use multi-party computation (MPC) techniques, which have been long studied within the field of cryptography. The goal of MPC is to allow a group of mutually distrusting parties to jointly evaluate a function of their private inputs, while leaking nothing but what can be deduced from the output of the function. Furthermore, protocols built on MPC techniques are generally secure, even if several parties (up to a fraction of the parties involved in the computation) collude to break the privacy of the other participants.

The traditional setting of MPC is one where the number of parties is relatively small and the network is assumed to be full mesh. Sadly, this precludes the immediate application of such techniques in the large, partial mesh networks which are prevalent today.

Recent approaches to monitoring in large networks employ an in-network paradigm [1] whereby monitoring is performed collaboratively by the network nodes themselves, using algorithms based on spanning trees [26,11] or gossiping [24,22]. For these applications, scalability is often taken to mean sub-linear growth in resource consumption growth in the size of the network.

Towards a general solution to the problem of collaborative network and security monitoring we present in this paper efficient protocols for computing sum, max, disjunction, and thresholds in partial mesh networks. These operations are sufficient to implement many of the aggregates of interest in monitoring. Our protocols are efficient, both in terms of message and computational overhead.

We focus in this paper on passive, "honest-but-curious" adversaries whereby attackers are bound to follow the protocol but may collude to learn information about the honest parties' inputs. This is much simpler than the active attack model also considered in multi-party computation and often leads to more efficient protocols. However, it is also a reasonable and attractive model in many practical situations where e.g. side conditions related to traffic observations and arguments of utility can be appealed to to ensure protocol behavior is adhered to.

The security of MPC protocols is commonly characterized by the size of collusions they remain secure against. Such thresholds become less meaningful for protocols, such as ours, which can be used on arbitrary networks. Thus, we analyze security in terms of tolerable adversary structures in the sense of Hirt and Maurer [20], and describe the tolerable structures in terms of graph theoretical properties of the network on which the protocol is executed.

As the need for monitoring is common to many areas, and our protocols are efficient, we believe there is a wide range of applications. We give a few examples of possible applications to set some context for the work.

Example 1 (Collaborative Security Monitoring). The need to aggregate security log information as part of general intelligence gathering is widely acknowledged, cf. [29]. The importance of collaboration is further emphasized by services such as Internet Storm Centre's `www.dshield.org`, where firewall logs can be shared, and aggregate statistics are collected.

Network providers and supervisors have strong interest in accurate security log aggregates, as this will allow more precise estimations of the global security situation, in order to take countermeasures and improve operations. There are, however, important privacy concerns, as log data, even in sanitized form, can reveal significant amounts of critical information concerning internal business and network operations. Previous work has explored techniques such as log anonymization and randomized alert routing to deal with this problem [29,25]. We argue that private aggregation techniques can be used in this scenario to produce practical security aggregates with strong privacy guarantees in near real time.

One application would be to collect aggregate packet- or flow counts to various destination ports. Due to the computational efficiency of our protocols, they could be run directly on network devices such as routers, and without the need to trust a third party.

Example 2 (Anonymous and robust peer-to-peer networks). Consider a peer-to-peer network for anonymous publication and retrieval of files where the network acts as a distributed storage. In this scenario, it could be of interest to compute the number of copies of a file to discover if further duplication of that file is needed, something that could be done by a private computation of a sum. It may also be useful to be able to query for availability of a file without learning any other information than if the file exists in the network or not, which would correspond to a private computation of disjunction.

Another application within the realm of peer-to-peer networking would be to implement monitoring of the overlay to enhance quality and research. This could be useful both for overlays with strict anonymity requirements, but also for more traditional file-sharing applications where individual users may still be hesitant to share information on e.g. the amount of data they've uploaded.

Example 3 (Joint control of SCADA systems). A research topic of growing importance is the security of Supervisory Control and Data Acquisition (SCADA) systems, e.g. systems controlling criticial infrastructure such as the electrical grid. Many different entities are involved in running the electrical grid, and they must co-operate to ensure production and consumption is balanced throughout the grid. However, many of the entities are direct competitors, which can prevent collaboration that would involve sharing of business-sensitive data.

Our protocols could be applied to monitor aggregate power flows over various areas of the grid, which is a summation. They could also be applied to compute the disjunction of alert statuses at operators. Then, if one operator has some form of disruptions, other operators would automatically be put on alert and be prepared in case the failure condition affects other parts of the grid. This would decrease the risk of cascading failures by giving early warnings to other operators, without sharing detailed information on the reliability of any individial operator.

We believe that in the scenarios presented above, the assumption of a passive adversary could be reasonable. For network monitoring, there is little to be gained for the participants in disrupting the computation of the aggregated information. In the P2P scenario, attacking monitoring is likely to be uninteresting, but searches and functions ensuring replication may be suitable candidates for protocols with stronger security properties, depending on the nature of the network. In the SCADA scenario, in addition to the small gains from actively manipulating the computations, it's possible that legislation would demand that data was retained for auditing, thus increasing the risk involved in cheating.

1.1 Our Contributions

Firstly, we give a protocol for summation, where we perform a single round of communication to achieve privacy, and then reduce the problem to non-private summation. A single group element is sent in each direction over every link in this extra round. The protocol is similar to a protocol by Chor and Kushilevitz [10], but adapted to a partial mesh network, and with a precise characterization of tolerable adversary structures. It is also similar to the dining cryptographers networks proposed by Chaum [9] which is essentially the same protocol but applied to provide sender untraceability.

Secondly, we present a computationally secure protocol for computing disjunction, based on homomorphic cryptosystem, such as El Gamal [15]. The protocol requires two rounds of communication and then uses a non-private protocol for summation. Computationally, it requires a small number of encryptions and decryptions per neighbor.

We also give a composition structure where the information-theoretically secure protocol for summation is composed with a standard protocol for computing some other function. We show that this can be used for several standard functions in network management, such as disjunction, min/max, or threshold detection. For this composition, there needs to be a complete subgraph K of the network such that no union of two sets from the adversary structure contains K. This is a reasonable assumption in many network monitoring applications where the members of K represent trusted servers appointed by a disjoint collection of network providers. This is similar to the use of trusted aggregation servers in [5,13,7]. The composition essentially performs an efficient and secure "aggregation" of all inputs to some smaller subset of parties who can then run a more expensive protocol with stricter connectivity requirements.

1.2 Related Work

There are general results [18,3] showing that every computable function can be privately evaluated in a multi-party setting, but the protocols involved require a full mesh network between the parties and can be prohibitively expensive to execute.

There are many specialized protocols for computing specific functions in the literature, that are more efficient than the general constructions. Examples of such protocols include an information-theoretically secure protocol for summation by Chor and Kushilevitz [10], and computationally secure protocols for disjunction and maximum by Brandt [6], which uses the homomorphic El Gamal cryptosystem as a building block. While such protocols are more efficient than the general solutions, they are still not scalable in the sense of the previous section. Just sending one message between every pair of parties forces each party to process too many messages.

In most of the works on multi-party computation, the parties are connected in a full mesh network. An article by Franklin and Yung [14] describes how to emulate the missing private channels between parties, and using their construction, protocols built for full mesh networks may also be run on arbitrary networks. However, this emulation can be very expensive, and may not always be possible, depending on what parties an adversary can corrupt.

There has also been research exploring how the network connectivity affects what functions can be computed with information-theoretical privacy. There are results due to Bläser *et al.* [4] and Beimel [2] categorizing the functions that can be computed on 1-connected networks.

The Dining Cryptographers problem, and its solution were discussed by Chaum [9]. They study the problem of creating a channel such that the sender of messages is untraceable and their suggested protocol is similar to our protocol for summation.

A technique that can be applied to sidestep the connectivity and performance issues of traditional MPC solutions is to aggregate data to a small set of semi-trusted nodes, who can then perform the computation. As these servers are few, it is more feasible to connect them with a full mesh network. Examples of such schemes include Sharemind [5], SEPIA [7], and a system by Duan and Canny [13]. These are similar to the protocols we present in Section 5, with a difference being that our protocols perform aggregation while collecting information from the nodes, thus decreasing the load on the servers performing the computation, but limiting what can be computed.

A number of authors propose additive secret sharing to secure information aggregation in large networks or databases. Privacy schemes similar to the sum protocol used here have been explored in the area of sensor networks and data mining [28,19]. In fact, a very large range of algorithms used in data mining and machine learning, including all algorithms in the statistical query framework [23], can be expressed in a form compatible with additive secret sharing. Several authors have investigated secure aggregation schemes for the case of a centralized aggregator node (cf. [21,27]). A solution with better scalability properties is proposed by Chan *et al.* [8]. There, an additive tree-based aggregation framework is augmented by hash signatures and authenticated broadcast to ensure that, assuming the underlying aggregation tree is already secured, an attacker is unable to induce an honest participant to accept an aggregate which could not be obtained by direct injection of some private data value at the attacking node. Other recent work with similar scope uses Flajolet-Martin sketches for secure counting and random sampling [16].

1.3 Organization of This Paper

We begin by presenting the security and computational model and various definitions in Section 2. We then proceed to outline and prove properties of the protocol for computing sums in Section 3. Then, we give a computationally secure protocol for computing disjunctions in Section 4. We then show a composition structure where the protocol for summation is composed with standard protocols to compute for instance disjunction in Section 5.

2 Model and Definitions

We consider multi-party computation (MPC) protocols for n parties, $P_1, \ldots, P_n$, and denote the set of all parties by $\mathcal{P}$. Each party P_i holds a private input, x_i, and the vector of all inputs is denoted x. The network is modeled as an undirected graph $\mathcal{G} = (\mathcal{P}, \mathcal{E})$ where messages can only be sent between adjacent parties.

For a graph $\mathcal{G} = (\mathcal{P}, \mathcal{E})$, we say that $\mathcal{G}$ is *disconnected* if there exists a pair of vertices such that there is no path between them. For a set of vertices $X \subseteq \mathcal{P}$, we denote by $\mathcal{G} - X$ the subgraph of $\mathcal{G}$ induced by the set of vertices $\mathcal{P} \backslash X$. In other words, $\mathcal{G} - X$ is the graph obtained by deleting all vertices in X and their incident edges from $\mathcal{G}$.

Definition 1 (Separator, set of vertices). *Given a graph $\mathcal{G} = (\mathcal{P}, \mathcal{E})$, a set of vertices $X \subseteq \mathcal{P}$ is called a* separator *of $\mathcal{G}$ if the graph $\mathcal{G} - X$ is disconnected.*

2.1 Adversary Structures

The most common adversary considered in the MPC literature is a threshold adversary corrupting up to a threshold of the parties. More generally, we can allow an adversary corrupting some subset of parties as specified by an *adversary structure* [20].

An adversary structure $\mathcal{Z}$ over $\mathcal{P}$ is a subset of the power set of $\mathcal{P}$, containing all possible sets of parties which an adversary may corrupt. We require that an adversary structure is monotone, i.e., it is closed under taking subsets.

Definition 2 (Separator, adversary structure). *In a network $\mathcal{G} = (\mathcal{P}, \mathcal{E})$, an adversary structure $\mathcal{Z}$ is called a* separator *of $\mathcal{G}$ if some element in $\mathcal{Z}$ is a separator of $\mathcal{G}$.*

From the monotonicity of $\mathcal{Z}$, it follows that if $\mathcal{Z}$ is not a separator of $\mathcal{G}$, then no matter what subset in $\mathcal{Z}$ the adversary chooses to corrupt, every corrupted party will have at least one honest neighbor. More precisely, for every set $C \in \mathcal{Z}$ it must be the case that every party $P \in C$ has at least one neighbor who is not in C. This observation is important for the proof of security of the computationally private protocol for disjunction given in Section 4.

2.2 Security Definition

The security definition of a multi-party computation says that the adversary should not learn anything from the protocol execution except what it can deduce from its inputs and the output of the function the protocol computes.

In the security analysis of our protocols, we only consider passive (*honest-but-curious*), static adversaries in a network with private and reliable channels. The protocols in Sections 3 and 5 are information-theoretically private, and the protocol in Section 4 is computationally private.

We consider information about the network the protocol is executed on to be public knowledge. Our protocols do not depend on honest parties knowing the network structure, but neither do anything to hide that information from the adversary.

We refer to [3,17] for details on security definitions for information-theoretical and computational security of multi-party computation.

2.3 Homomorphic Cryptosystems

A cryptosystem $\mathsf{CS} = (\mathsf{Gen}, \mathsf{E}, \mathsf{D})$ is said to be homomorphic if the following holds.

- Each public key pk output by Gen defines groups of messages, randomness, and ciphertexts, denoted $\mathcal{M}_{pk}$, $\mathcal{R}_{pk}$, $\mathcal{C}_{pk}$ respectively, for which the group operations are efficiently computable.
- For every public key pk, every messages $m_1, m_2 \in \mathcal{M}_{pk}$, and every $r_1, r_2 \in \mathcal{R}_{pk}$: $\mathsf{E}_{pk}(m_1, r_1)\mathsf{E}_{pk}(m_2, r_2) = \mathsf{E}_{pk}(m_1 + m_2, r_1 + r_2)$.

It is convenient in our applications to use additive notation for both the group of messages and the group of randomness. However, we do not require that the cryptosystem is “additively homomorphic”, e.g., that $\mathcal{M}_{pk} = \mathbb{Z}_m$ for some from integer m. Thus, any homomorphic cryptosystem with sufficiently large message space suffices, e.g., El Gamal. We remark that we do not use the fact that the cryptosystem is homomorphic over the randomness.

3 Computing Sums

We present an information-theoretically secure protocol for computing sums over a finite Abelian group. The protocol is similar to a protocol by Chor and Kushilevitz [10], but adapted to arbitrary networks, and with a precise characterization of tolerable adversary structures. It is also similar to a protocol by Chaum [9], with the difference that we explicitly create shared random secrets by a straightfoward technique and use the protocol for summation rather than sender-untraceability.

When computing sums, privacy comes cheap. We can take any non-private protocol for sums, `NonPrivateSum`$(x_1, \ldots, x_n)$, and augment it with a single additional round to turn it into a private protocol. The protocol admits all adversary structures $\mathcal{Z}$ which do not separate the network $\mathcal{G}$. This requirement on the adversary structure is necessary in the information-theoretical setting.

Protocol 1 (Sum). In the protocol for computing $\sum_{i=1}^{n} x_i$ over an Abelian group $\mathcal{M}$, on the network $\mathcal{G} = (\mathcal{P}, \mathcal{E})$, $P_i \in \mathcal{P}$ proceeds as follows:

1. For each neighbor P_j, pick $r_{i,j} \in \mathcal{M}$ randomly and send it to P_j.
2. Wait for $r_{j,i}$ from each neighbor P_j.
3. Compute $s_i = x_i - \sum_{(P_i,P_j)\in\mathcal{E}} r_{i,j} + \sum_{(P_i,P_j)\in\mathcal{E}} r_{j,i}$.
4. Output $\texttt{NonPrivateSum}(s_1, \ldots, s_n)$.

We begin by observing that the protocol correctly computes the sum of the inputs x_i. For every value $r_{i,j}$ sent in step 1 of the protocol, that value is added to s_j and subtracted from s_i, so all $r_{i,j}$ cancel when summing the s_i.

We now show that the protocol is information-theoretically private with respect to passive, static adversaries. We do this by showing that for any non-separating collusion, the remaining s_i values are uniformly random, conditioned on $\sum_{i=1}^{n} s_i = \sum_{i=1}^{n} x_i$.

Theorem 1. *Protocol 1 is information-theoretically private to a passive and static adversary if the adversary structure $\mathcal{Z}$ does not separate the network $\mathcal{G} = (\mathcal{P}, \mathcal{E})$.*

To prove the theorem, we begin by stating a lemma from which the theorem follows immediately.

Lemma 1. *Consider executions of Protocol 1 on a network $\mathcal{G} = (\mathcal{P}, \mathcal{E})$ where: the output $\sum_{i=1}^{n} x_i$, a non-separating collusion $\mathcal{C}$, and the inputs x_i and communication $r_{i,j}, r_{j,i}, s_i$ for $P_i \in \mathcal{C}$ are fixed. For such executions the remaining values s_i for $P_i \in \mathcal{P}\backslash\mathcal{C}$ are uniformly random, conditioned on $\sum_{i=1}^{n} s_i = \sum_{i=1}^{n} x_i$.*

Proof (Theorem 1). The values $r_{i,j}$ sent in the first round are independent of the input. By Lemma 1, for any fixed input and random tapes of a non-separating collusion, and fixed output of the protocol, the remaining messages have the same distribution. $\square$

Proof (Lemma 1). Consider two vectors $s = (s_1, \ldots, s_n)$, $s' = (s'_1, \ldots, s'_n)$, and two vectors of inputs $x = (x_1, \ldots, x_n), x' = (x'_1, \ldots, x'_n)$ such that $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} x'_i = \sum_{i=1}^{n} s_i = \sum_{i=1}^{n} s'_i$, and $s_i = s'_i, x_i = x'_i$ for all $P_i \in \mathcal{C}$.

Let R denote an $n \times n$ matrix of $r_{i,j}$, where $r_{i,j} = 0$ if (P_i, P_j) is not an edge in $\mathcal{G}$. Define $s(x, R)$ to be the vector of s_i values sent in the protocol when executed on input x with random values R. The value at the ith position of $s(x, R)$ is denoted by $s_i(x, R)$.

We show that the probability of s being sent on input x is equal to the probability of s' being sent on input x'. This is done by, for any tuple of vectors s, s', x, x' constructing a bijective function $f(R)$ such that if $s = s(x, R)$ then $s' = s(x', f(R))$. The function $f(R)$ has the form $f(R) = R + R'$ for an $n \times n$ matrix $R' = (r'_{i,j})_{i,j}$. Furthermore, $r'_{i,j} = 0$ if $P_i \in \mathcal{C}$ or $P_j \in \mathcal{C}$.

We note that $s = s(x, R)$ holds iff R is such that for each P_i we have $s_i - x_i = \sum_{j=1}^{n} r_{j,i} - r_{i,j}$. Thus, for R' we need precisely that for each P_i we have

$$\sum_{j=1}^{n}(r'_{j,i} - r'_{i,j}) = (s'_i - x'_i) - (s_i - x_i)\,. \tag{1}$$

Since $\mathcal{C}$ is not a separator, there exists a directed spanning tree T that spans the honest parties, $\mathcal{P}\backslash\mathcal{C}$. Let $r'_{i,j} = 0$ if (P_i, P_j) is not an edge in T. We can now fill in R' iteratively during a postorder traversal of T. When a non-root P_i is visited, only $r'_{j,i}$ for its parent P_j is still undefined on the ith row and column of R', and its value is determined by Equation 1.

When the root is visited, R' is completely filled in and we know that Equation 1 holds for all other parties. Consider the sum of Equation 1 over all parties. The left hand side satisfies $\sum_{i=1}^{n}\sum_{j=1}^{n}(r'_{j,i} - r'_{i,j}) = 0$. The right hand side also satisfies $\sum_{i=1}^{n}(s'_i - x'_i) - (s_i - x_i) = 0$ since $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} x'_i = \sum_{i=1}^{n} s_i = \sum_{i=1}^{n} s'_i$. Since Equation 1 holds for all parties except for the root, it must also hold for the root. □

We would like to remark that the proof of Lemma 1 does not make use of the monotonicity of the adversary structure $\mathcal{Z}$. Thus, if we allow non-monotone adversary structures (for instance, if parties 1 and 2 must always be corrupted jointly), the protocol is still private given that $\mathcal{Z}$ does not separate the network $\mathcal{G}$.

It is intuitively clear that sums cannot be privately computed if $\mathcal{Z}$ separates the network, and this is indeed the case. In [2], Beimel gives a characterization of the functions that can be privately computed in non-2-connected networks, with an adversary structure consisting of all singleton sets, and shows that sums cannot be computed in that setting. Any information-theoretically private protocol computing sums tolerating $\mathcal{Z}$ separating the network can be turned into a protocol violating the bounds given in [2] by standard simulation techniques, and cannot exist.

4 A Computationally Secure Protocol For Disjunction

We now consider the problem of computing the disjunction of all parties' inputs, and present a computationally secure protocol, requiring two rounds of communication and an execution of non-private protocol for summation.

As a building block, we need a cryptosystem $\mathsf{CS} = (\mathsf{Gen}, \mathsf{E}, \mathsf{D})$ that is homomorphic. We further need that the group of messages $\mathcal{M}_{pk}$ is the same group for all keys generated with the same security parameter, κ. For notational convenience, we denote this group $\mathcal{M}$. We require the cryptosystem to have IND-CPA security, i.e., resistance to chosen-plaintext attacks. We relax the correctness requirements slightly, and allow our protocol to incorrectly output `false` with negligible probability $2^{-\kappa}$.

In this protocol, we construct a linear secret sharing of a group element which is zero if all the parties' inputs are `false`, and a uniformly random group element otherwise. The protocol then proceeds by opening the share, which is done by (non-private) summation.

Conceptually, each party contributes either a zero or a random group element, depending on its input. However, it is important that a party does not know the group element representing its own input, as this would allow it to recognize if it was the only party with input `true`. In order to achieve this, we apply homomorphic encryption to allow its neighbors to jointly select how its input is represented.

If the security requirements are relaxed slightly, and it is acceptable that the adversary can learn if any other parties had input `true`, then Protocol 1 can be used instead (with each party herself choosing 0 or a random element as her input).

For ease of notation, we identify `false` with 0, and `true` with 1. In the description of the protocol, we abuse notation slightly and multiply a value by a party's input as a shorthand for including or excluding terms of a sum.

Protocol 2 (Disjunction). In the protocol for computing $\mathtt{Or}(x_1, \ldots, x_n)$, where $x_i \in \{0,1\}$, on the network $\mathcal{G} = (\mathcal{P}, \mathcal{E})$, based on a homomorphic cryptosystem $\mathsf{CS} = (\mathsf{Gen}, \mathsf{E}, \mathsf{D})$, $P_i \in \mathcal{P}$ proceeds as follows:

1. Generate a key-pair $(pk_i, sk_i) \leftarrow \mathsf{Gen}(1^\kappa)$.
2. For each neighbor P_j, pick a random element $a_{i,j} \in \mathcal{M}$, and send $pk_i, c_{i,j} = \mathsf{E}_{pk_i}(a_{i,j})$ to P_j.
3. Upon receiving $pk_j, c_{j,i}$ from P_j, pick a random $r_{i,j} \in \mathcal{M}$, and send $c'_{i,j} = \mathsf{E}_{pk_j}(r_{i,j}) + x_i c_{j,i}$ to P_j.
4. Wait for $c'_{j,i}$ to be received from every neighbor P_j, and then compute $s_i = \sum_{(P_i,P_j)\in\mathcal{E}} (\mathsf{D}_{sk_i}(c'_{j,i}) - r_{i,j})$
5. Compute $\mathtt{NonPrivateSum}(s_1, \ldots, s_n)$ and output 0 if the sum is the identity, and 1 otherwise.

The protocol is efficient, both in terms of computational resources and communication. Each party needs to perform two encryptions, one decryption and one ciphertext multiplication per neighbor. The first encryption does not depend on the input, and can be performed off-line. The communication overhead of the protocol is two rounds, in addition to performing a (non-private) summation.

Theorem 2. *Protocol 2 for computing the disjunction of n bits on a network $\mathcal{G} = (\mathcal{P}, \mathcal{E})$, gives the correct output if it is* `false`*, and gives an incorrect output with probability $2^{-\kappa}$ when the correct output is* `true`.

Proof. Consider the sum

$$\sum_{i=1}^{n} s_i = \sum_{i=1}^{n} \sum_{(P_i,P_j)\in\mathcal{E}} (x_j a_{i,j} + r_{j,i} - r_{i,j}) = \sum_{(P_i,P_j)\in\mathcal{E}} x_j a_{i,j} \, .$$

If all x_j are 0, clearly the sum is 0. Otherwise, it is a sum of uniformly random group elements, and thus has uniformly random distribution. In particular, with probability $1 - 2^{-\kappa}$ it is non-zero. □

4.1 Privacy

Theorem 3. *If the cryptosystem* CS *is* (t, ϵ)*-IND-CPA secure, then no adversary running in time* $t - t'$*, for a small* t'*, can violate the privacy of Protocol 2 with advantage more than* $\frac{n^2}{4}\epsilon$*.*

The proof of Theorem 3 begins like the proof of Theorem 1 with a combinatorial lemma similar to Lemma 1, essentially saying that unless the adversary learns something about the values $a_{i,j}$ from seeing them encrypted, it cannot violate the privacy of Protocol 2. Given the lemma, we apply a hybrid argument to prove the security of the protocol.

Lemma 2. *Consider executions of Protocol 2 on a connected network* $\mathcal{G} = (\mathcal{P}, \mathcal{E})$ *with input* x *such that* $x_i =$ `true` *for at least one* P_i*, and where a collusion* $\mathcal{C} \in \mathcal{Z}$ *from a non-separating adversary structure* $\mathcal{Z}$*, and communication* $a_{i,j}, r_{i,j}, r_{j,i}, s_i$ *for* $P_i \in \mathcal{C}$ *is fixed. For such executions, the values* s_i *for* $P_i \in \mathcal{P} \backslash \mathcal{C}$ *have a uniform and independent distribution.*

Proof (Theorem 3). We begin with the observation that if all the parties have input `false`, then the protocol behaves exactly as Protocol 1 with zeroes as inputs and by Lemma 1, then the honest parties' s_i will be uniformly random conditioned on $\sum_{i=1}^{n} s_i = 0$.

First, consider the case where the inputs of all corrupted parties are `false`. In this case, a simulator that independently samples the pk_i, $c_{i,j}$, $r_{i,j}$ and s_i included in the adversary's view, conditioned only on $\sum_{i=1}^{n} s_i = 0$ if the output is `false`, or $\sum_{i=1}^{n} s_i \neq 0$ otherwise perfectly simulates the protocol to the adversary, by the previous observation and Lemma 2. Thus, in this case, the adversary cannot violate the privacy of the protocol.

Now, consider the case when at least one of the corrupted parties has input `true`. We begin by constructing a simulator S_0 that randomly selects inputs and $a_{i,j}$ for all honest parties, conditioned on the output matching the output it should simulate. It then follows the protocol to simulate the adversary's view.

We now construct hybrid simulators, S_k, working like S_0 but replacing the first k ciphertexts $c_{i,j}$ in the adversary's view by random ciphertexts. It follows from the (t, ϵ)-IND-CPA security of $\mathsf{E}_{pk_i}(x)$ that no adversary running in time $t - t'$, for some small t' required to run the simulator S_k, can distinguish between the views simulated by S_k and S_{k-1}.

Assume that the adversary's view includes T ciphertexts $c_{i,j}$, so the view simulated by S_T contains no information on the $a_{i,j}$ sent by honest nodes to corrupted nodes. There can be at most $(n/2)^2$ edges between honest and corrupted nodes, so $T \leq n^2/4$. By Lemma 2, the distribution of simulated $r_{i,j}$ and s_i values is exactly the same as in a real execution, so the view simulated by S_T contains no information on the honest parties inputs. □

Proof (Lemma 2). Consider the following mental experiment, where we modify an execution of the protocol in two steps.

Modification 1. For each neighbor P_j of P_i we subtract $x_i c_{j,i}$ from $c'_{i,j}$ in Step 3 of the protocol and add $x_i a_{j,i}$ to s_i in Step 4 of the protocol. It is easy to see that this does not change the distribution of either s_i or $\mathsf{D}_{sk_i}(c'_{i,j})$ for any neighbor P_j.

Modification 2. Remove all encryptions and decryptions. This transforms Steps 3-5 of the protocol into an execution of Protocol 1, where P_i holds the input $\sum_{j=1}^{n} x_i a_{j,i}$.

From Lemma 1 we conclude that with the two modifications, the s_i are independently distributed conditioned on $\sum_{i=1}^{n} s_i = \sum_{i=1}^{n} \sum_{j=1}^{n} x_i a_{j,i}$, but the right side of this equation is randomly distributed when some $x_i = 1$ and $a_{i,j}$ for some neighbor P_j is randomly distributed. From the conditions of the lemma, we know there is at least one P_i such that $x_i = 1$, and from the monotonicity of $\mathcal{Z}$ and that it is non-separating, we know that every party has an honest neighbor. Thus, the s_i are uniformly and independently distributed. This concludes the proof. □

4.2 Computing the Maximum

In the setting with passive adversaries, it is easy to construct a protocol for computing the maximum by repetition and parallel composition of a protocol for disjunctions.

Assume the inputs are integers of ℓ bits. We can then compute the disjunction of the most significant bits of all parties' inputs, which is also the most significant bit of the maximum of the inputs. We then proceed to the next most significant bit. When a party learns that its input is smaller than the maximum (its input was 0 and the output was 1), it participates with input 0 in the remaining protocol executions.

Several bits can be handled in parallel to reduce the number of rounds at the cost of more protocol executions. To find the maximum of k bits, one can run $2^k - 1$ parallel disjunction computations, where the parties set their inputs based on if their k most significant bits represent an integer greater or equal to $2^k - 1, 2^k - 2, \ldots, 1$, respectively. Thus, to find the maximum of ℓ-bit integers, one can run $\lceil \ell/k \rceil$ rounds of protocols for disjunction, with $2^k - 1$ protocol executions in each round.

5 General Composition

Many functions can be computed as a function of the sum of inputs of the parties. Examples include disjunction, counting and threshold functions. In this context, a threshold function is a function returning `true` if the sum of inputs exceeds some threshold and `false` otherwise.

We can combine our Protocol 1 with standard protocols (which assume full mesh communications) to construct information-theoretically secure protocols for computing such functions. The benefit of this approach is that information-theoretical security is achieved in a partial mesh network while maintaining efficiency. Another approach would have been to simulate the missing edges (e.g., with the techniques from [14]) and then immediately using standard protocol, but this approach is generally more expensive in terms of communication.

By this composition, we essentially run a cheap protocol to "accumulate" the inputs of most parties and then let some small subset of parties run a more expensive protocol and jointly act as a trusted party. This can be useful when performing computations with a large number of parties where some subset can be trusted not to collude with each other. This can be compared to the trusted servers in [5,13,7].

Executing the standard protocol requires a complete network, so this construction is only applicable when $\mathcal{G}$ contains a subgraph K that is complete. Furthermore, tolerable adversary structures $\mathcal{Z}$ are those that do not separate the graph, and which, restricted to K, are tolerable by the standard protocol being used. For most protocols, the requirement will be that no two subsets in $\mathcal{Z}$ cover K, or using notation from [20], the predicate $Q^{(2)}(\mathcal{Z}|_K, K)$ must hold.

Protocol 1 constructs a secret sharing of the sum of the parties inputs and then opens it. When we adapt the protocol for composition, we only construct the secret sharing, and accumulate the sum (still shared) in the nodes in K.

As an example, we give an information-theoretically secure protocol for disjunction. Here, we let each party input 0 or 1 (for `false` and `true`) and then use a protocol by Damgård *et al.* [12] for comparison.

Protocol 3 (Disjunction). In the protocol for computing $\texttt{Or}(x_1, \ldots, x_n)$ where $x_i \in \{0,1\}$ on the network $\mathcal{G} = (\mathcal{P}, \mathcal{E})$ with a set $K \subseteq \mathcal{P}$ of designated parties, $P_i \in \mathcal{P}$ proceeds as follows:

1. For each neighbor P_j, pick $r_{i,j} \in \mathbb{Z}_p$ randomly and send it to P_j.
2. Wait for $r_{j,i}$ from each neighbor P_j.
3. Compute $s_i = x_i - \sum_{(P_i,P_j)\in\mathcal{E}} r_{i,j} + \sum_{(P_i,P_j)\in\mathcal{E}} r_{j,i}$.
4. Compute $s = \sum_{P_j \notin K} s_j$ using `NonPrivateSum`.
5. If in K, execute comparison protocol from [12] to test if $s + \sum_{P_j \in K} s_j = 0$.

Theorem 4. *Protocol 3 is information-theoretically private to a passive and static adversary if the adversary structure $\mathcal{Z}$ does not separate the network $\mathcal{G} = (\mathcal{P}, \mathcal{E})$ and there is a complete subgraph $K \subseteq \mathcal{G}$ such that no two sets in $\mathcal{Z}$ cover K.*

Proof. The values $r_{i,j}$ are independent of the input. By the restriction on $\mathcal{Z}$ there must be at least one party in K not corrupted by the adversary. By Lemma 1 we know that the s_i values input to `NonPrivateSum` are uniform and independent. Thus, the adversary gains no information from these, and by the composition theorem [17, Theorem 7.5.7], we conclude that the protocol is private. □

6 Conclusion

In this paper we have given efficient protocols for privately evaluating summation and disjunction on any network topology. The ability to privately evaluate these two basic primitives have applications in several widely varying contexts. As the most expensive part of our protocols is the task of non-private summation, privacy comes very cheaply.

We believe that the question of which functions can be efficiently privately evaluated in arbitrary network topologies is an interesting topic for further study.

References

1. The FP7 4WARD project. `http://www.4ward-project.eu/`.
2. Amos Beimel. On private computation in incomplete networks. *Distributed Computing*, 19(3):237–252, 2007.
3. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10. ACM, 1988.
4. Markus Bläser, Andreas Jakoby, Maciej Liskiewicz, and Bodo Manthey. Private computation: k-connected versus 1-connected networks. *J. Cryptology*, 19(3):341–357, 2006.
5. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.
6. Felix Brandt. Efficient cryptographic protocol design based on distributed el gamal encryption. In Dongho Won and Seungjoo Kim, editors, *ICISC*, volume 3935 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2005.
7. Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *19th USENIX Security Symposium*, Washington, DC, USA, August 2010.
8. Haowen Chan, Adrian Perrig, and Dawn Xiaodong Song. Secure hierarchical in-network aggregation in sensor networks. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 278–287. ACM, 2006.
9. David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptology*, 1(1):65–75, 1988.
10. Benny Chor and Eyal Kushilevitz. A communication-privacy tradeoff for modular addition. *Inf. Process. Lett.*, 45(4):205–210, 1993.
11. M. Dam and R. Stadler. A generic protocol for network state aggregation. In *Proc. Radiovetenskap och Kommunikation (RVK)*, 2005.
12. Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.
13. Yitao Duan and John F. Canny. Practical private computation and zero-knowledge tools for privacy-preserving distributed data mining. In *SDM*, pages 265–276. SIAM, 2008.

14. Matthew K. Franklin and Moti Yung. Secure hypergraphs: Privacy from partial broadcast. *SIAM J. Discrete Math.*, 18(3):437–450, 2004.
15. Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.
16. Minos N. Garofalakis, Joseph M. Hellerstein, and Petros Maniatis. Proof sketches: Verifiable in-network aggregation. In *ICDE*, pages 996–1005. IEEE, 2007.
17. Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.
18. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229. ACM, 1987.
19. Wenbo He, Xue Liu, Hoang Nguyen, Klara Nahrstedt, and Tarek F. Abdelzaher. PDA: Privacy-preserving data aggregation in wireless sensor networks. In *INFOCOM*, pages 2045–2053. IEEE, 2007.
20. Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *J. Cryptology*, 13(1):31–60, 2000.
21. Lingxuan Hu and David Evans. Secure aggregation for wireless networks. In *Workshop on Security and Assurance in Ad hoc Networks*, page 384. IEEE Computer Society, 2003.
22. Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, 2005.
23. Michael J. Kearns. Efficient noise-tolerant learning from statistical queries. In *STOC*, pages 392–401, 1993.
24. David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *FOCS*, pages 482–491. IEEE Computer Society, 2003.
25. Patrick Lincoln, Phillip A. Porras, and Vitaly Shmatikov. Privacy-preserving sharing and correlation of security alerts. In *USENIX Security Symposium*, pages 239–254. USENIX, 2004.
26. Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.
27. Bartosz Przydatek, Dawn Xiaodong Song, and Adrian Perrig. SIA: secure information aggregation in sensor networks. In Ian F. Akyildiz, Deborah Estrin, David E. Culler, and Mani B. Srivastava, editors, *SenSys*, pages 255–265. ACM, 2003.
28. Matthew Roughan and Yin Zhang. Secure distributed data-mining and its application to large-scale network measurements. *SIGCOMM Comput. Commun. Rev.*, 36(1):7–14, 2006.
29. Adam J. Slagell and William Yurcik. Sharing computer network logs for security and privacy: A motivation for new methodologies of anonymization. *CoRR*, cs.CR/0409005, 2004.

V

Timing is Everything — the Importance of History Detection

Gunnar Kreitz

KTH – Royal Institute of Technology
`gkreitz@kth.se`

Abstract. Browsers have long admitted an attack allowing a malicious web page to detect whether the browser has visited a target web site by using CSS to style visited links and read out the style applied to a link. For a long time, this CSS history detection attack was perceived as having small impact. Lately, highly efficient implementations of the attack has enabled malicious web sites to extract large amounts of information. Following this, browser developers have begun to deploy measures to protect against the attack.
In this work, we demonstrate that the impact of history detection is not limited to attacks on a user's privacy. We demonstrate an attack where history detection is used to time the execution of a *Flow Stealing* attack, redirecting the victim's browser at a particularly vulnerable times in transaction flows. This chaining together of two attacks highlights the importance of finally closing the long-standing history detection security hole.
We also show that our Flow Stealing attack can be applied without using the CSS history stealing attack if the attacker can intercept the victim's network traffic. Noting that different browsers place different restrictions on cross-frame navigation through JavaScript window handles, we suggest a stricter policy based on pop-up blockers to prevent Flow Stealing attacks.

Keywords. CSS History Detection, Flow Stealing, Cross-site Request Forgery

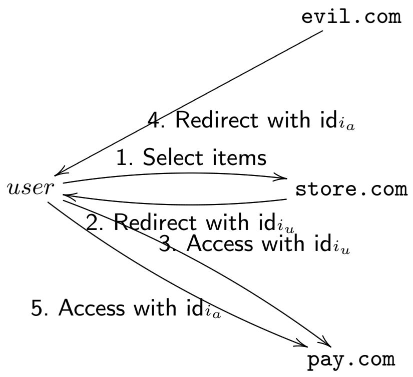
1 Introduction

Cross-site request forgery (CSRF) and Session fixation are two large classes of attacks against web pages, with both attacks meriting high placement on the OWASP Top 10 list 2010 [1]. Both types of attacks are well documented, and there are many proposed counter-measures.

One special form of CSRF is the login CSRF attack, as highlighted by Barth *et al.* [2]. In a login CSRF attack, the attacker logs the victim on to a legitimate site using an account controlled by the attacker. The purpose of this is for the attacker to extract or use information stored by the victim's activity on the site. Examples of such abuse includes stealing the search history of the victim, or using stored credit card details to transfer money or make purchases.

As discussed in [2], the login CSRF attack is an example of vulnerabilities in session initialization. Another type of vulnerability in the same class is that of session fixation, where the attacker tricks the victim into logging in on a legitimate site with a session identifier known to the attacker. The attacker can then visit the legitimate site using the same session identifier and then be logged in as the victim.

To protect against this class of session initialization attacks, a number of different methods have been proposed. These include employing a validation token, looking at the `Referer` [sic] header, using custom HTTP headers, looking at the `Origin` header, and generating a new session identifier at critical points such as login.



evil.com
4. Redirect with id_{i_a}
1. Select items
user
store.com
2. Redirect with id_{i_u}
3. Access with id_{i_u}
5. Access with id_{i_a}
pay.com

Fig. 1. Basic attack flow overview

Most of the protection mechanisms are primarily concerned with protecting the user's flow on a single site. What happens when we consider state transfer that occurs between two different domains? In particular, we look at the handover from a store to a payment provider. A typical integration mechanism is that the retailer sends information about the purchase to the payment provider (at least the total amount to be paid) and gets a transaction identifier. The store then sends the user to the payment provider with the transaction identifier, either by a GET or POST request[1]. In this paper, we will outline an attack where the adversary at this point redirects the victim's browser to the same payment provider, but with a different transaction id. We illustrate the flow in Figure 1. We refer to this as *Flow Stealing*.

Two questions arise: firstly, how does the attacker know when to redirect the victim's browser, and secondly, how does the attacker redirect the browser?

[1] Several payment provider also provide lightweight integration where the store directly redirects the customer with information about the purchase instead of a transaction identifier. This does not materially affect the attack, so we consider this equivalent to sending a transaction identifier.

Our attacker can make use of an old and well-known security hole, CSS history detection [3], in order to time her attack. To be able to redirect the victim's browser, the attacker needs JavaScript running in the browser and a window handle to the window which is to be redirected. This can be achieved for instance if the visitor visits the attacker's web page and clicks on a link opening in a new tab.

While the CSS history detection attack is well-known, to the best of our knowledge it has not previously been used to time another attack. The security hole has been regarded mostly as a privacy leak, and despite the fact that it has been publicly documented since at least 2000, mainstream browsers started to deploy protections only in 2009. Firefox's stable version is still vulnerable, but the upcoming version 4 closes the hole. Internet Explorer only just closed the hole in version 9, a version not available on Windows XP.

1.1 Attacker and Victim Model

In this work we consider two forms of attackers: an attacker running a web page, and an attacker who can perform network attacks against the victim's network connection to other, legitimate sites.

Our primary focus will be on an attacker operating a web site, `evil.com`, and through various means entices victims to visit her page. We furthermore assume that the attacker can convince victims to click on a link from `evil.com`, and use the opened tab to buy something. Our envisioned attacker could make some money (legally) by hosting advertisements or participating in an affiliate program. Thus, a potential attacker is an affiliate site gone rogue. We remark that our attacker is *weaker* compared to the traditional attacker model in many CSRF and XSS attacks, as the attacker only needs the user to follow a *legitimate* link to a well-known site.

We consider a potential victim of our attack who follows the guidelines taught by the security community. She will only give sensitive information over https, but not before verifying that the certificate is authentic. Her machine does not have any malware or spyware installed on it, and her browser is fully patched. In addition to a security-conscious victim, we assume that the attacked flow is on domains served only over https.

We also consider a *network attacker* who can intercept and modify the victim's network traffic. There are several ways in which an attacker could get this ability. For an attacker on the same local network as the victim, the attacker can utilize standard tools such as ARP or DHCP spoofing to get access to the victim's traffic. An attacker more interested in large-scale attacks could set up a Tor exit node and thereby become a man-in-the-middle against anonymous victims. Given man-in-the-middle access to the user's network communication, all information sent and received over http can trivially be attacked, but our focus is on pages protected by https, which is intended to protect also against network attacks. We do not assume that the network attacker can trick the victim into visiting her web site and clicking a link there, so the network attacker is not strictly stronger than our normal attacker.

1.2 Our Contribution

In this paper, we describe a new type of attack which we call Flow Stealing. Our attack makes new use of a well-known security issue in the CSS specification to time the execution of a redirection attack. By timing the redirection precisely, the attacker can give the user a false sense of security by having her browse well-known sites before the attack is executed. This new use of an old attack emphasizes the importance of closing also minor security holes where the impact is not fully understood. While it seems that most major browsers have closed, or will close, the CSS history detection hole our attack makes use of in upcoming releases, we also consider a network-based attacker which can make use of encrypted network traffic from the victim instead of the CSS hole.

Our attack furthermore highlights a part of typical web flows which is difficult to protect using current protection mechanisms, namely legitimate cross-domain redirects.

We identify several scenarios in which the attack can be mounted, and we suggest new protection mechanisms which can be used to prevent our attack, as well as similar attacks. In particular, we point out the dangers of allowing JavaScript to navigate and close windows to which it holds a window handle and propose a new policy based on pop-up blocking.

1.3 Outline and Running Example

As a running example, we will consider an attacker attacking the flow from `store.com` to `pay.com`. The following is the outline of the basic attack:

1. User visits `evil.com`, and follows link to `store.com`
2. User interacts with `store.com`, eventually reaching checkout
3. `store.com` creates transaction on `pay.com`, which assigns transaction identifier i_u
4. `store.com` redirects user to `pay.com` with transaction identifier i_u.
5. `evil.com` detects that user hits `pay.com`
6. `evil.com` creates transaction on `pay.com`, which assigns transaction identifier i_a
7. `evil.com` redirects the user's tab to `pay.com` with transaction identifier i_a

We remark that several variations on this attack are conceivable. For instance, the attacker may choose to register a domain name which is visually similar to that of the real payment provider and redirect the victim to her domain, tricking her into giving up credit card information. Such attacks have been discussed in the literature in conjunction with phishing, c.f. [4]. However, we believe that a victim will be much less likely to notice minor differences in a domain name when following a link from a well-known store than when following a link from a phishing email.

From the description, we notice that the attacker requires both the ability to detect when the user reaches `pay.com`, as well as the ability to redirect her tab once she is there. We begin by discussing how the redirect is accomplished in

Section 2. We then proceed to discuss the issue of timing the redirect in Section 3. After this, we discuss which browsers are currently affected in Section 4 and report on our experiences with a proof-of-concept implementation. Then, we discuss some counter-measures and recommendations in Section 5. Finally, we conclude and briefly discuss future work in Section 6.

2 Redirecting the User Tab

How can the attacker redirect the victim's browser? Firstly, this requires the attacker to get the victim's browser to run some malicious JavaScript. This is easily accomplished for an attacker who convinces the victim to visit `evil.com`, as the page can contain the JavaScript required for the attack. A network attacker using a man-in-the-middle attack can insert malicious JavaScript into any page or script content served over unprotected http. For more details, see Section 2.2

Apart from having JavaScript running in the victim's browser, the script needs to have a window handle to the tab in which the user is visiting `store.com`, and later `pay.com`. If the victim opened the tab by clicking a link on `evil.com`, the attacker's JavaScript can store a window handle to the tab. We defer discussion of the man-in-the-middle case to Section 2.2.

Many browsers permit JavaScript to freely navigate any top-level window handles it holds. Browsers also used to be liberal with navigation of frames, something which posed a security hole as discussed by Barth *et al.* [5]. However, when switching to a stricter window navigation policy for frames and iframes, the new policy was not applied to top-level frame navigation. A motivation for this may have been that the impact of allowing an attacker to navigate top-level frames is smaller, as the user can look at, and verify the address bar. However, as our attack demonstrates, this does not always help, for instance when the attacker only changes an opaque transaction id in the URL.

One notable exception is Opera which does not allow a window w_1 to navigate a window w_2 to which it has a handle if w_2 is currently browsed to a https page at a domain different from w_1. There is a simple way for our attack to get around this restriction in Opera, but it does make the attack easier to detect for the victim. We discuss the circumvention in Section 2.1.

As Opera has had this restriction for a long time, we believe that few sites in practice perform this site of navigation windows that they have opened. Thus, it is unlikely that making the window navigation policy via JavaScript handles more restrictive would incur any huge compatibility penalties. However, doing so would make our flow stealing attack more difficult, so there are security benefits from adopting a restrictive window navigation policy also of top-level frames. We propose a new policy in Section 5.2.

We remark that once an attacker's JavaScript has a window handle to a window, it retains its rights over that window regardless of what happens. In particular, a user manually typing in a different address in the navigation bar does not revoke any of the opener's privileges.

2.1 Working Around Opera's Navigation Restrictions

Opera prevents a window from navigating another window via a window handle in some scenarios. In our flow stealing attack, we need to change the address of the victim's window when it goes to `pay.com`, which we assume is served over https. Thus, we propose a slightly different variation when attacking the Opera browser.

If a window w_1 wants to navigate the window w_2 to some address, it can accomplish a similar effect which may not be noticed if it closes window w_2 and navigates itself to the address it wanted window w_2 to go to. We are not aware of any browser placing restrictions on closing windows via a JavaScript handle. Depending on the user configuration and how many tabs the user typically has open, this "navigation" may be more or less noticeable.

If the attacker can close window w_2, why not simply open another window with the right address in its place? The answer is that such an attempt will likely be prevented by a pop-up blocker. All mainstream browsers today prevent sites from arbitrarily opening new tabs, unless the action is initiated by a user action such as a mouse click.

2.2 Page Modification by a Network Attacker

In our attacker model, we consider a network attacker who is not assumed to be able to entice victims to visit her web site and click on links. This way, the web attacker needs some other way to get JavaScript running in the victim's handle, as well as a window handle to a window where the victim then makes a purchase.

Most web browsing today is done over http, instead of https. However, we assume that both `store.com` and `pay.com` have invested in security and are served only over https. Thus, the network attacker cannot perform man-in-the-middle attacks against these domains directly.

Our network attacker can easily modify any other page the victim visits over http. Thus, an attacker could write a proxy inserting malicious JavaScript into all pages the victim visits over http. To make this attack efficient, we assume that the attacker wants to adapt the JavaScript as little as possible to the page the attack is inserted into.

We begin with a discussion on what the JavaScript should do. We assume that the network attacker wants to avoid detection, and thus not modify any user-visible behavior of web sites. This means that she will want to insert JavaScript on the page such that it captures a window reference to any window opened by the page. A page can be opened for one of two reasons, either by the user clicking on a link with the `target` attribute set to "`_blank`", or by JavaScript on the page calling `window.open`.

Thus, the attack flow for our network attacker is as follows:

1. Victim visits `http://example.com`
2. Attacker's proxy inserts JavaScript into returned `example.com` page

3. Victim clicks on link to `example2.com` opening in new window
4. Attacker's JavaScript captures a reference to the opened window

in which situation the network attacker is almost in the same position as when the victim visits `evil.com` and follows a link from there.

We start with links using the `target` attribute to open a new window. The attacker can insert JavaScript which executes when the page is loaded, and which loops through all anchor tags on the web page. When it reaches an anchor tag with `target` set to `_blank`, it modifies the tag to call a JavaScript function opening the window and storing the window handle when clicked. We remark that as these tags are easily detectable if the attacker parses the page, it would be easy to make this modification statically as part of a man-in-the middle attack as well. We illustrate a simplified JavaScript example in Figure 2.

Handling windows opened by JavaScript on the original web site at first appears more difficult. To detect when windows may be opened could involve dynamic analysis of JavaScript code. However, there is an easy way to capture references opened by JavaScript on the original page.

```
window.real_open = window.open;
window.open = function(URL, name, specs, replace) {
   var openedWindow = real_open.apply(this, arguments);
   storeReferenceAndStartTiming(openedWindow);
   return openedWindow;
}

function modifyLinks() {
   var links = document.getElementsByTagName("a");
   for (i=0; i<links.length; i++) {
      if(links[i].getAttribute("target") == "_blank") {
         links[i].setAttribute("onClick", "window.open(\"" +
links[i].getAttribute("href") + "\"); return false;");
      }
   }
}
window.onload = modifyLinks;
```

Fig. 2. Simplified JavaScript code to capture window references from non-malicious pages

To do this, we use a technique which has been used by Phung *et al.* [6] to construct a security mechanism for policy enforcement in JavaScript. The technique is based on the observation that even built-in functions can be aliased by user-defined functions in JavaScript. Thus, the malicious JavaScript can replace the `window.open` method with a JavaScript function which calls the original `window.open` method and stores a copy of the returned window handle before returning it to the caller. Slightly simplified JavaScript code illustrating the principle is shown in Figure 2.

3 Timing the attack

We now turn to the question of how the attacker can learn when the user is redirected to `pay.com`. We present two mechanisms for accomplishing this. The first, and easiest method builds on the well-known CSS history detection attack to periodically poll whether the `pay.com` URL has become visited. Secondly, we also propose a method for the network attacker where she by monitoring the network traffic can time her attack.

3.1 CSS History Detection

An early feature in web browsers is the distinction between a visited and an unvisited link. With the advent of Cascading Style Sheets (CSS), the creator of a web site gained the ability to decide how the two types of links would be rendered. It was soon realized [7] that this feature could be abused by a web site to determine of its visitor had also visited some other site. The CSS 2.1 specification [8, Section 5.11.2] notes the vulnerability and states that browsers may treat all links as unvisited or implement other counter-measures.

We remark that while an attacker can test if a visitor has visited a specific URL, she cannot extract the full browsing history of the visitor. In particular, she does not learn anything about URLs she cannot guess. The rate at which the attacker can test URLs is also an issue as it limits the privacy exposure of the attack. Here, the increasing prevalence of Web 2.0 applications and the accompanying optimization in general JavaScript performance has benefited an attacker. Speeds of 30000 tested URLs/second have been reported by Janc and Olejnik [3] with their optimized version of the attack.

Recall that the integration with a payment provider is typically done by setting up a transaction and then redirecting the user to the payment provider with a unique transaction identifier assigned by the payment provider. The attacker is not able to predict the transaction identifier, so if it had been a part of the URL, the attacker would not be able to use the CSS history detection attack to learn when the user visited the payment provider. However, common practice is to send the transaction identifier to the payment provider as a POST parameter to a static URL, which allows our attack to work.

History detection attacks have been studied in the academic literature, and several demonstration web sites [9,10] have been created to raise awareness of the issue. Wondracek *et al.* [11] showed that stolen history data can also be used for a de-anonymization attack against users of social network sites. Jakobsson and Stamm [12] discussed the potential of using history detection in phishing attacks. Benevolent uses of the history detection attack have also been discussed. One example is to guess at which OpenID provider a user has to ease OpenID-logins [13], and another is to detect if a user has visited malicious sites and may have had malware installed [14].

The threat to user privacy is the most well-known implication of history detection. When coupled with fast testing, a non-trivial part of the user's visiting patterns can be extracted. This allows for testing of URLs containing location information such as zip codes entered on e.g., weather sites. In their real-world experiment Janc and Olejnik [3] noted that they could detect the US zip code for 9.2% of tested users.

3.2 Using History Detection to Learn When the User Reaches a Page

In our application of the history detection attack, we are not interested in the user's browsing history but rather in what the user is currently doing. In particular, we want to know when the user's current browsing session reaches a target page (e.g., the landing page of a payment provider). To accomplish this, we can use the history detection attack to frequently poll and thus determining when the status of the target page changes from unvisited to visited.

This use of history detection requires that the target page is marked as unvisited in the browser when the attack is started. Thus, the attack is easier to perform the quicker the browser forgets about visited links, in total contrast to privacy attacks which typically benefit from longer history retention. The CSS specification leaves it up to the implementor to select for how long a link will be treated as visited, and the large browsers have selected different periods. Internet Explorer and Safari stores history for 20 days, and Firefox for 90 days. Opera does not limit the time, but rather limits the number of stored entries to 1000. Chrome does not remove visited status, except when explicitly requested by the user.

Thus, our flow stealing attack is best suited to attacking pages which users trust, but which they visit rarely. We believe that payment providers, such as Paypal, fall in this category for many users.

3.3 Limitations of CSS History Detection

There are several ways in which the victim can be protected from the way we use CSS history detection in this attack. Firstly, Baron [15] has proposed a mechanism to close the CSS history detection security hole. The most basic mechanism involved is that the data returned by the JavaScript `getComputedStyle` method always return data as if the link had been unvisited. Furthermore, it prevents visited status of link from affecting which pictures are loaded, the layout of the page, and the time it takes to render a page to prevent a number of side-channel attacks. This proposal (or similar defenses) has been implemented in Firefox 4[2], Internet Explorer 9, as well as in browsers based on the WebKit rendering engine, such as Chrome and Safari. This means that in the upcoming versions, most mainstream browsers will have closed the CSS history stealing hole. Users

[2] At the time of this writing, Firefox 4 is still in beta, and the latest stable version is still vulnerable

may not always be able to upgrade to the latest version, for various reasons. For instance, Internet Explorer 9 is not supported on Windows XP, which will prevent many users from upgrading. Also, even if they can, some users simply refuse to upgrade their browsers.

There are some mechanism a user can deploy to protect herself, apart from switching or upgrading their browser. A user may choose to configure their browser not to store any browsing history. However, this comes at a usability price. Firefox users may also choose to install the SafeHistory extension [16] which essentially applies the same-origin policy to visited status on links, only treating a link as visited if it has been visited by a link from the current domain.

CSS history detection is not the only history detection attack that has been proposed against web browsers. In [17], Felten and Schneider discuss timing attacks to determine if cacheable elements of pages are present in the victim's cache. However, such attacks are less suitable to our history detection usage where we are not interested if the victim has historically visited a site, but rather in detecting the moment in time when a specific page is visited. To the best of our knowledge, cache timing attacks cause the tested object to be cached, and thus the same object cannot be tested twice, making the attack unsuitable for repeated polling. We remark that there is a companion extension to SafeHistory called SafeCache [16] to protect against cache timing attacks.

3.4 Network Based Timing

In the case of a network attacker who has access to the user's network traffic, there are alternative timing mechanism for the cases when the CSS history detection timing mechanism does not work. As we assume that all the victim's browsing of `store.com` and `pay.com` is via https, making the attacker unable to directly observe what the victim browses at the target domains. However, https does not attempt to protect against an attacker learning that the victim is visiting a certain domain, or the sizes of the request and response.

There are several ways for the network attacker to learn when the victim visits `pay.com`. The first is by simply observing the victim's DNS traffic. When the attacker sees the victim's computer performing a DNS lookup for the IP address of `pay.com`, she can assume that the victim's browser is going to request something from that domain. However, if the victim frequently visits `pay.com`, she may already have the IP address cached in her browser, and thus not issue a DNS lookup when visiting the domain again. Another mechanism for the attacker is to look up the IP addresses of servers for `pay.com` and then trigger the attack when she sees the victim's computer connecting to one of those IP addresses over the https port. We refer to Section 3.4 for a discussion on how the attacker communicates that the attack is to trigger to the JavaScript running in the victim's browser.

Both these mechanisms may trigger the attack too early if other pages include elements from the `pay.com` domain, for instance if `store.com` includes a `pay.com` logo on their payment page. While this type of logo inclusion does occur, we remark that it is common practice for stores to host payment logos on their own servers, or for static content such as logos to be hosted on separate domains.

The attacker can learn if the store features `pay.com` logos served directly by `pay.com` servers by simply visiting the store herself before beginning the attack. If this is the case, she can perform a more thorough flow inspection and instead of just looking for a connection establishment to the right IP and port, analyze the number of bytes sent in each direction and the number of connections made to distinguish between the victim fetching a logo and visiting the landing page at the payment provider.

Communicating Back to Victim's Browser When discussing the alternate timing mechanism available to the network attacker, we stated that the attacker "triggers the attack". However, the attacker is located as a man-in-the-middle to the victim's network traffic, and to trigger the attack, she must activate code running as JavaScript in a tab in the victim's browser. How is the trigger information communicated back to the victim's browser?

We first remark that in our network attacker scenario, the malicious JavaScript has been inserted by the attacker on a web page not controlled by the attacker. Thus, the malicious JavaScript is prevented by the same-origin policy from directly communicating with the attacker-controlled server at `evil.com` via convenient mechanisms such as XMLHttpRequest.

However, as the attacker is mounting a man-in-the-middle attack on the victim's network traffic, this problem can be circumvented by the attacker intercepting and responding to requests to some specific path, regardless of what host the path is supposed to be located at. This allows the JavaScript inserted by the attacker to use XMLHttpRequest to periodically send a request to a long path which the attacker will intercept. The attacker will not forward such requests, but instead respond with a boolean value indicating if the flow stealing redirect should be activated. There are also other options available, such as periodically loading images from `evil.com` and using the size of the returned images as a one-way communication channel to the JavaScript running in the victim's browser.

4 Impact and Feasibility of Flow Stealing

We have now described our proposed flow stealing attack, showing how it can be performed both by an attacker operating a web site as well as by a network attacker who can intercept the user's network traffic. Apart from the conditions imposed by the type of attacker, the feasibility of the attack also depends on the victim's browser. In particular, some browsers have implemented a proposed protection [15] against the CSS history detection attack that we propose as a timing mechanism in flow stealing.

4.1 Browser Features

Our flow stealing attack combines two different vulnerabilities. Firstly, the attacker must be able to monitor when the victim is directed to `pay.com`. The primary mechanism for accomplishing this is by using a well-known history detection hole. Secondly, the attacker must at that point in time redirect the victim to `pay.com` with a new transaction id.

While the redirection part is crucial to the flow stealing attack, the CSS history detection vulnerability is not needed for network attackers. As discussed in Section 3.4, there is an alternate timing mechanism which can be utilized by the network attacker.

All mainstream browsers allow the redirection part of our attack. However, on the Opera browser, the attacker cannot simply redirect the victim's tab, but must instead close the tab and redirect another tab as discussed Section 2.1. This makes the attack more noticeable, as an alert victim may notice that a tab closed and become suspicious and abort the transaction.

Table 1. Summary of browser's susceptibility to flow stealing.

Browser	CSS History Detection	Window Navigation
Firefox 3.6.15	Yes	Permissive
Firefox 4RC	No	Permissive
IE 8.0.7600.16385	Yes	Permissive
IE 9.0.8112.16421	No	Permissive
Chrome 10.0.648.151	No	Permissive
Safari 5.0.4	No	Permissive
Opera 11.01	Yes	Restricted

To explore the feasibility of our attack, we have tested the latest versions of browsers to see if the classic CSS-based history detection attack works, and what restriction they place on cross-domain window navigation through window handles. We present our results in Table 1. In the table, "CSS History Detection" indicates if the CSS history detection attack works. Redirection indicates if a window handle can always be redirected via JavaScript ("Permissive") or not ("Restricted"). The browsers were tested on Windows 7. We do not believe any of the results depend on the operating system the browser is run on.

4.2 Experiences with a Proof-of-Concept

In addition to testing the individual pieces of our flow stealing attack, we have also developed a proof-of-concept implementation of the attack as performed by a web-site hosting attacker. We consider the simplest version of attack which can be performed with a static html containing JavaScript for the attack using the CSS history detection timing mechanism. In our proof-of-concept, we replaced `store.com` with the donation page of a charity, to simplify testing (the donation page of the charity contains a link directly to the payment provider).

In our proof-of-concept, the transaction set up by the attacker has the attacker as the recipient instead of the charity. The recipient information is displayed by the payment provider, so an alert user could notice that their flow had been hijacked by an attacker. To reduce the risk of this, an attacker could register names with the payment provider which are similar to the stores or charities that she will attack.

Guessing the Price To make the attack convincing to the victim, the attacker needs to set up a transaction with the exact same cost that the user expected. While we believe users may not always check security indicators on web pages, we think that a large fraction of users would notice if the payment provider listed a different price compared to the store. We have not implemented any techniques for creating a transaction with the correct price in our proof-of-concept.

There are several ways for an attacker to guess the price. The easiest way is to attack subscription services or stores which sell a specific item or service for a fixed price, or a small number of different options so that the attacker can simply guess at the most common price. One such example is online streaming services such as Hulu, Napster, Netflix, and Spotify.

For stores with larger inventories, the attacker can use the CSS history detection attack to determine what items the victim has browsed and/or put in her shopping basket, depending on the URL scheme employed by the attacked store.

5 Proposed Counter-Measures

In this section, we discuss a simple server-side defense against CSS history detection that can be applied by payment providers for their landing page. We proceed to discuss the problem of frame navigation as it applies to top-level frames and propose a new policy based on pop-up blocking. Finally, we discuss why traditional CSRF defenses do not protect against flow stealing.

We note that our attack uses JavaScript to perform the redirection attack, so users can protect themselves against flow stealing by disabling JavaScript. However, this does remove functionality from a large number of web sites, so most users are unlikely to do so.

5.1 Closing the CSS History Detection Hole

We are happy that most of the mainstream browsers appear to be closing the CSS history detection hole in upcoming versions. By closing this hole, attackers are denied the easiest route for performing flow stealing attacks. However, for various reasons, users are not always able to upgrade to the newest version of software in a timely manner. To protect users which are not able to upgrade, we propose that high-profile sites such as payment providers should consider implementing a server-side defense.

While landing pages of payment providers are external URLs in the nomenclature of [12], they could apply a protection technique by recommending sites linking to them to insert a random number in the link, which is simply ignored by the payment provider. As most payment providers want to help stores to very easily integrate payments, standard practice seems to be to provide some static HTML code to be included on the store's web site. Such code could include JavaScript code to generate a random number in the browser which is inserted into the URL of the landing page in a way that is ignored by the payment provider. This would prevent the link from being guessable, and thus detectable via CSS history detection.

We hope that by demonstrating that the impact of the CSS history stealing attack extends beyond previously documented privacy attacks, users, if made aware of flow stealing attacks, may be more inclined to upgrade their browsers or employ other protection measures. We believe that browser developers should consider patching the security hole also in older versions of their browsers. In particular, this applies to Internet Explorer, as the hole was patched in IE 9 which is not available for versions of Windows before Vista.

5.2 Limiting Window Manipulation via Window Handles

There is a difference in policy between browsers on what limits are applied to how a page can change the URL of another window to which it has a JavaScript window handle. Opera restricts such navigation based on the current location of the frame, and protects frames navigated to https sites from being navigated from another window. In Chrome, Firefox, Internet Explorer, and Safari, the opener is allowed to freely navigate an opened window, and in some of them, also other windows apart from the opener.

Frame navigation has previously been showed as being dangerously permissive in the context of embedded frames and iframes by Barth *et al.* [5], which influenced browser developers to implement a more restrictive policy. They note that top-level frames are often exempt from the browser's frame navigation policy, and that top-level frames are less vulnerable as their URL is shown in the location bar.

While it is true that top-level frames are less vulnerable than embedded frames, there is still a danger in permissive policies for navigation of top-level frames. We cannot trust a user to, at every point in time in their browsing session, validate that the location in the location bar is correct. For instance, we cannot expect users to note if their location is changed to a similarly looking URL, or identical looking URL via a homograph attack. Neither can we expect users to notice if opaque identifiers in sessions are replaced.

The fact that different policies have been implemented in different browsers indicates that it is unlikely that a large number of pages rely on the most permissive policies for their functionality. The only policy restricting our flow stealing attack is Opera's. However, as we discussed in Section 2.1, Opera's policy is still sufficiently permissive that it allows flow stealing attacks by closing the window and redirecting the window running the attacking JavaScript. Thus, we argue that a replacement policy should not only restrict navigation, but rather all actions affecting the window, including closing it and resizing it (an attacker could emulate closing by resizing to a very small size).

We are not aware of any important applications where a window w_1 needs to modify another window w_2 where the modification is not prompted by user interaction with window w_1. For what types of user interaction would a user expect w_1 to modify the state of another window w_2? We argue that in any user interaction that would not allow w_1 to open a new window, w_1 should not be allowed to modify the state of another window either. In mainstream browsers today, the situations in which w_1 is allowed to open a window is restricted by a *pop-up blocker*. We believe a user would not expect w_1 to modify any windows unrelated to it, a policy already implemented in the Firefox browser which limits navigation to the *opener* window.

As far as we know, each mainstream browser implements their own algorithm for pop-up blockers, and that the pop-up blocker is enabled by default. Thus, most web sites have been adapted to page manipulations allowed by the pop-up blocking policies of browsers. We are not aware of any detailed descriptions of pop-up blocking algorithms, but they appear to work satisfactorily in major browsers. According to Chen [18], browser developers are hesitant to specify the exact policies used as that may prevent them from modifying the policy later, if a loophole is discovered.

Thus, we propose the following policy for controlling a window via a window handle:

Policy 1 (Window navigation, Proposed) *A window w_1 can modify (e.g., navigate, close, or resize) another window w_2 only if it is the opener of w_2, and the pop-up blocker policy currently allows w_1 to open a window.*

Furthermore, we believe that rights to a window should be relinquished entirely if the user manually navigates (e.g., by entering a new URL in the address bar) the tab. Currently, this does not appear to affect the rights granted to the JavaScript holding the handle to the window, but we believe it would match the user's expectation more closely that the opener retains no special privileges if the user navigates the window.

5.3 Traditional CSRF Defenses do not Prevent Flow Stealing

Our flow stealing attack has some similarities to traditional CSRF attacks, so one may wonder if traditional CSRF defenses protect against flow stealing as well. Sadly, the answer is no, and new techniques are needed to protect against flow stealing. We briefly mention the two major CSRF defenses from the literature and discuss why they do not protect against flow stealing. At the core of the problem is that in flow stealing the victim's browser is redirected at a point in time where the control is passed between sites operated by two different entities, a store and a payment provider. Thus, the hijacked request is legitimately a cross-site request.

The most common class of CSRF defense consists of a secret validation token that must be sent along with all state-modifying requests, and that is matched to the user's session. There are several different implementations of this technique, and there are some subtleties in implementing the protection correctly, c.f. [2]. Such tokens are designed to protect flows internally on web-sites, and are not immediately applicable to cross-site flows.

A second technique is based on inspecting either the `Referer` or the `Origin` HTTP header. Typically, this is described as only allowing requests if the host in the header matches the current host, but the policy could easily be extended to allowing external requests from some specific set of domains. As an example, a payment provider may require that users making payments to `store.com` come to `pay.com` with an Origin header set to `store.com`. However, this does not prevent flow stealing, as the attacker can register as a merchant with the payment provider and redirect via the correct domain for that merchant. The attacker can also redirect the victim to a fake payment site instead of the legitimate site, thus bypassing any controls that could be implemented by a payment provider.

6 Conclusion and Future Work

In conclusion, we have demonstrated an attack on current web browser implementations. The attack uses the CSS history detection attack, which has been publicly documented for about a decade, to time a redirection attack. By redirecting the tab the victim is using at a point where the victim legitimately expects to perform some security critical action, the victim can be tricked into doing something more sensitive than what can be achieved by e.g. phishing. We hope that our attack further aids in demonstrating the importance of closing the CSS history detection hole, and that all mainstream browsers will adopt counter-measures.

Our core attack uses on two issues in the victim's browser: history detection and the ability for JavaScript opening a tab to later navigate the opened tab. Many modern browsers today have closed at least one of these holes, but the attack still works in the current stable version of Firefox. To prevent the attack from re-appearing via alternative timing mechanisms or new history detection holes, we propose a new policy for window manipulation based on pop-up blockers.

As future work, we propose developing a proof-of-concept version of the network attack as well. The purpose of such a proof-of-concept prototype would be to show that while closing the CSS history detection hole is an important step, it is also important to further limit JavaScript cross-site frame navigation, as well as deploying https as a default for a larger fraction of Internet sites. We note that other proof-of-concept attacks such as Firesheep [19] have been able to quickly raise public awareness of security issues and caused deployment improvements at large sites.

References

1. OWASP: OWASP top 10 - 2010 (2010) http://owasptop10.googlecode.com/files/OWASP\%20Top\%2010\%20-\%202010.pdf.
2. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. In Ning, P., Syverson, P.F., Jha, S., eds.: ACM Conference on Computer and Communications Security, ACM (2008) 75–88
3. Janc, A., Olejnik, L.: Web browser history detection as a real-world privacy threat. In Gritzalis, D., Preneel, B., Theoharidou, M., eds.: ESORICS. Volume 6345 of Lecture Notes in Computer Science., Springer (2010) 215–231
4. Holgers, T., Watson, D.E., Gribble, S.D.: Cutting through the confusion: A measurement study of homograph attacks. In: USENIX Annual Technical Conference, General Track, USENIX (2006) 261–266
5. Barth, A., Jackson, C., Mitchell, J.C.: Securing frame communication in browsers. Commun. ACM **52**(6) (2009) 83–91
6. Phung, P.H., Sands, D., Chudnov, A.: Lightweight self-protecting javascript. In Li, W., Susilo, W., Tupakula, U.K., Safavi-Naini, R., Varadharajan, V., eds.: ASIACCS, ACM (2009) 47–60
7. Ruderman, J.: Bug 57351 - css on a:visited can load an image and/or reveal if visitor been to a site https://bugzilla.mozilla.org/show_bug.cgi?id=57351.
8. W3C: Cascading style sheets level 2 revision 1 (CSS 2.1) specification http://www.w3.org/TR/CSS2/.
9. Anonymous: Did you watch porn (2010) http://www.didyouwatchporn.com/.
10. Janc, A., Olejnik, L.: What the internet knows about you (2010) http://www.wtikay.com/.
11. Wondracek, G., Holz, T., Kirda, E., Kruegel, C.: A practical attack to de-anonymize social network users. In: IEEE Symposium on Security and Privacy, IEEE Computer Society (2010) 223–238
12. Jakobsson, M., Stamm, S.: Invasive browser sniffing and countermeasures. In Carr, L., Roure, D.D., Iyengar, A., Goble, C.A., Dahlin, M., eds.: WWW, ACM (2006) 523–532
13. Kennedy, N.: Sniff browser history for improved user experience (2008) http://www.niallkennedy.com/blog/2008/02/browser-history-sniff.html.
14. Jakobsson, M., Juels, A., Ratkiewicz, J.: Remote harm-diagnostics http://www.ravenwhite.com/files/rhd.pdf.
15. Baron, L.D.: Preventing attacks on a user's history through CSS :visited selectors http://dbaron.org/mozilla/visited-privacy.
16. Jackson, C., Barth, A., Bortz, A., Shao, W., Boneh, D.: Protecting browsers from DNS rebinding attacks. In Ning, P., di Vimercati, S.D.C., Syverson, P.F., eds.: ACM Conference on Computer and Communications Security, ACM (2007) 421–431
17. Felten, E.W., Schneider, M.A.: Timing attacks on web privacy. In: ACM Conference on Computer and Communications Security. (2000) 25–32
18. Chen, R.: The internet explorer pop-up blocker follows guidelines, not rules http://blogs.msdn.com/b/oldnewthing/archive/2007/08/31/4656351.aspx.
19. Butler, E.: Firesheep http://codebutler.com/firesheep.

VI

Lower Bounds for Subset Cover Based Broadcast Encryption

Per Austrin[⋆] and Gunnar Kreitz

KTH – Royal Institute of Technology, Stockholm, Sweden
{austrin,gkreitz}@kth.se

Abstract. In this paper, we prove lower bounds for a large class of Subset Cover schemes (including all existing schemes based on pseudo-random sequence generators). In particular, we show that

– For small r, bandwidth is $\Omega(r)$
– For some r, bandwidth is $\Omega(n/\log(s))$
– For large r, bandwidth is $n - r$

where n is the number of users, r is the number of revoked users, and s is the space required per user.

These bounds are all tight in the sense that they match known constructions up to small constants.

Keywords: Broadcast Encryption, Subset Cover, key revocation, lower bounds.

1 Introduction

A Broadcast Encryption scheme is a cryptographic construction allowing a trusted sender to efficiently and securely broadcast information to a dynamically changing group of users over an untrusted network. The area is well studied and there are numerous applications, such as pay-per-view TV, CD/DVD content protection, and secure group communication. For instance, the new Advanced Access Content System (AACS) standard, which is used for content protection with next-generation video disks, employs Broadcast Encryption.

A Broadcast Encryption scheme begins with an initialization phase where every user is given a set of secrets. Depending on the application, a "user" in the scheme could be an individual, a subscriber module for a cable TV receiver, or a model of HD-DVD players. When the initialization is complete, the sender can transmit messages. For each message it wants to transmit, it selects a subset of users to receive the message. We will refer to this subset of intended recipients as *members* (another common name is the privileged set). It then encrypts and broadcasts the message, using the secrets of the members, in such a way that only the members can decrypt the broadcast. Even if all the non-members (or *revoked* users) collude, they should not be able to decrypt the broadcast. The term key revocation scheme is also used for Broadcast Encryption schemes.

[⋆] Research funded by Swedish Research Council Project Number 50394001.

The performance of a Broadcast Encryption scheme is generally measured in three parameters: bandwidth, space and time. Bandwidth is the size of the transmission *overhead* incurred by the scheme, space is the amount of storage for each user, and time is a measurement of the computation time needed for users to decrypt a message. In this paper, we will focus on the tradeoff between bandwidth and space.

In general, Broadcast Encryption schemes work by distributing a fresh *message key*, so that only the current members can recover the message key. The actual message is then encrypted under the message key and broadcast. This construction means that the bandwidth, i.e., the overhead incurred by the scheme, does not depend on the sizes of the messages the sender wants to transmit.

The problem of Broadcast Encryption was first described by Berkovits in [5], and later Fiat and Naor started a more formal study of the subject [7].

There are two naive schemes solving the broadcast encryption problem. In the first naive scheme, we give each user her own secret key shared with the sender. With this scheme, the space is 1 and the bandwidth is m, where m the number of members. In the second naive scheme, we assign a key to every possible subset of users, and give all users belonging to a subset access to the key for that subset. In this case the space is 2^{n-1}, where n is the number of users, and the bandwidth is 1.

In 2001, the *Subset Cover* framework was introduced by Naor *et al.* [16], along with two schemes, Complete Subtree and Subset Difference. In Subset Cover based schemes, there is a family of subsets of users, where each subset is associated with a key. When the sender wishes to make a broadcast, she finds a *cover* of the members using the subsets and encrypts the message key with each of the subset keys used in the cover. Both naive schemes can be seen as Subset Cover schemes; in the scheme with constant space, the family consists only of singleton subsets, and in the scheme with constant bandwidth, the family consists of all subsets of users. The Subset Cover principle is very general, and most published schemes are Subset Cover schemes.

In most Subset Cover schemes, each user is a member of a large number of subsets, so storing the key for each subset would be expensive in terms of memory. To solve this, the keys are chosen in such a way that users can compute the keys they should have using some smaller set of secrets and a *key derivation algorithm*. Schemes where keys are unrelated are called *information-theoretic*.

The most common key derivation algorithm is a straightforward application of a Pseudo-Random Sequence Generator (PRSG). The first protocol of this type was Subset Difference which has a bandwidth of $\min(2r-1, n-r)$ with a user space $s = \mathcal{O}(\log^2 n)$ where r is the number of revoked users. Many more schemes [11,4,10,13,12] have been proposed using the same kind of key derivation. They all have a bandwidth of $\mathcal{O}(r)$, the same as Subset Difference, and their improvements lie in that some of them have a space of $\mathcal{O}(\log n)$, some offer increased flexibility, and some improve the bandwidth to $c \cdot r$ for $c < 1$ (as opposed to $c = 2$ in the original scheme).

Other forms of key derivation, such as RSA accumulators [2,3,8] and bilinear pairings [6], have also been studied for Subset Cover based broadcast encryption.

There have been attempts to reduce bandwidth by modifying the problem, for instance by allowing some *free-riders* (non-members who can still decrypt the broadcast) [1] or relaxing the security requirements [14].

There has been some analysis of lower bounds for Broadcast Encryption schemes. In 1998 Luby and Staddon [15] showed $s \geq \left(\frac{\binom{n}{r}^{1/b}}{b} - 1\right)/r$ for Broadcast Encryption without key derivation, using the Sunflower lemma. This bound was sharpened in 2006 by Gentry *et al.* [9] to $s \geq (\binom{n}{r}^{1/b} - 1)/r$. We remark that schemes using key derivation beat these bounds.

1.1 Our Contribution

Table 1. Upper and lower bounds for Subset Cover schemes

Key derivation	Lower bound	Assumption	Upper bound	Space
None	$r\frac{\log(n/r)}{\log(rs)}$	—	$r\log(n/r)$	$s = \mathcal{O}(\log n)$
PRSG, small r	$\Omega(r)$ (**our**)	$s \leq \text{poly}(n)$	$\mathcal{O}(r)$	$s = \mathcal{O}(\log n)$
PRSG, worst r	$\Omega\left(\frac{n}{\log s}\right)$ (**our**)	—	$\mathcal{O}\left(\frac{n}{\log s}\right)$	—
PRSG, large r	$n - r$ (**our**)	$r \geq n - \frac{n}{6s}$	$n - r$	$s = \mathcal{O}(1)$

We present lower bounds for a large class of Subset Cover schemes, including existing schemes based on PRSGs. These lower bounds match known constructions up to a small constant, showing that current PRSG-based schemes are essentially optimal. Table 1 gives a summary of our results.

Our bounds on the bandwidth usage are strong, and show that the early Subset Difference scheme is in fact very close to being optimal. For instance, our bound for small r shows that improving the bandwidth to $o(r)$ would require super-polynomial space, which is unreasonable. In fact, depending on the application, space is generally considered reasonable if it is at most logarithmic (or possibly polylogarithmic) in n.

Our second result implies that, in order to get constant bandwidth b, the space required is exponential. It also implies that, using polylogarithmic space, the worst case bandwidth will be almost linear, $n/\log\log n$.

The third result says that, for a small number of members, the first naive scheme is optimal. With polylogarithmic amount of space, this holds even if the number of members is almost linear, $n/\text{poly}\log n$.

Also, in most current schemes, the decryption time for members is limited to be polylogarithmic in n. Our proofs do not make use of any such restrictions, so allowing longer decryption time than current schemes cannot lower the bandwidth requirements.

1.2 Organization of This Paper

In Section 2 we discuss the structure of Subset Cover based Broadcast Encryption schemes and define the class of schemes, Unique Predecessor schemes, for which we prove lower bounds. In Section 3 we give a proof showing that with polynomial memory in clients, the bandwidth consumption is $\Omega(r)$ for "small" r. In Section 4, we prove a bound for generic r, and in particular show that for $r \approx \frac{n}{e}$, the bandwidth is at least $\frac{n}{1.89 \log s}$. Section 5 shows that for a large number of revoked users, the worst case bandwidth is $n - r$, i.e., the same as for the naive scheme where every user has a single key.

2 Preliminiaries

In this section, we review some preliminaries. The concepts of Broadcast Encryption and Subset Cover schemes are described, and notation will be introduced. We also define a class of Broadcast Encryption schemes called Unique Predecessor (UP) schemes to which our lower bounds apply.

2.1 Broadcast Encryption

In Broadcast Encryption, we have a trusted sender, and a set of users. After some initialization, the sender can securely broadcast messages to some subset of the users in a way which is efficient for both the sender and users. We will refer to the users who are targeted by a broadcast as *members* and the users who are not as *revoked* users. As the name Broadcast Encryption implies, we assume there is a single broadcast medium, so all users see the messages transmitted by the sender.

When evaluating the efficiency, three parameters are measured: bandwidth, space, and time (for decryption). Most Broadcast Encryption schemes transmit encrypted keys, so we will measure the bandwidth in terms of the number of encrypted keys to be transmitted. The sender uses the broadcast encryption to distribute a message key K_m and then encrypts the actual message under K_m, so the bandwidth overhead incurred does not depend on the size of the actual message.

The bandwidth required for a scheme with n users out of which r are revoked can, and generally will, vary, depending on which r users are revoked. We define the bandwidth $b = f(n, r)$ of the Subset Cover scheme as being that of the maximum bandwidth over the choice of the set of revoked users $R \subseteq [n]$ such that $|R| = r$. Thus, when we say that the bandwidth is at least $c_1 r$ for $r \leq n^{c_2}$, we mean that for every such r, there is at least one choice of r revoked users which requires bandwidth at least $c_1 r$.

Similarly, we measure the space as the number of keys, seeds, or other secrets that a user must store to be able to correctly decrypt transmissions she should be able to decrypt. It need not be the case that all users have to store the same amount of secrets, so we let the space of a scheme be the size of the largest amount of secrets any one user must store.

We remark that, in general, the keys and secrets may vary in length, so that our convention of simply counting the number of keys may not measure the exact bandwidth or space. However, such differences are generally small and not taking them into account costs us at most a small constant factor.

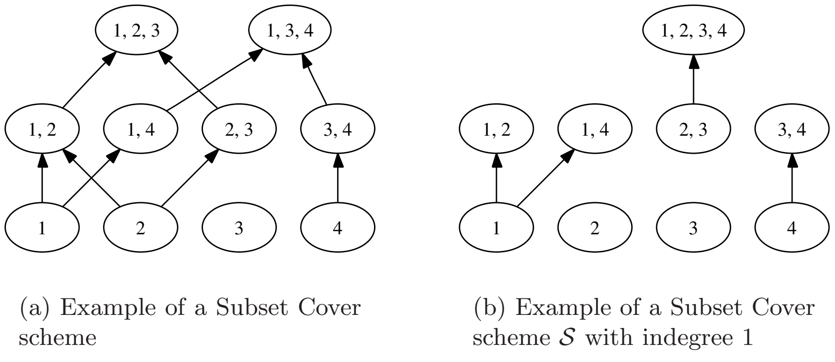
In this paper, we will not concern ourselves with the computational time of the clients. Our only assumption will be the very natural (and necessary) assumption that users cannot derive keys which they should not have access to.

Broadcast Encryption schemes can be classified as either stateful or stateless. In a stateful scheme, a transmission from the sender may update the set of secrets a user uses to decrypt future broadcasts, whereas in the stateless case, the secrets are given to the user at initialization and then remain constant. We focus on the largest family of Broadcast Encryption schemes, Subset Cover schemes, and such schemes are stateless.

2.2 Notation

Throughout the paper, we will use the following notation. We let n denote the total number of users, and identify the set of users with $[n] = \{1, \ldots, n\}$. We let m denote the number of members and r the number of revoked users (so $n = r + m$). The space of a scheme is denoted by s and the bandwidth by b. Note that we are generally interested in the bandwidth as a function of r (or equivalently, of m).

2.3 Subset Cover Schemes



(a) Example of a Subset Cover scheme

(b) Example of a Subset Cover scheme $\mathcal{S}$ with indegree 1

Fig. 1. Illustration of Subset Cover schemes

In this paper, we consider a family of Broadcast Encryption schemes known as Subset Cover schemes, introduced in [16]. In a Subset Cover scheme, the sender starts by creating a family of subsets of users. Each such subset is associated with a key. To make an encrypted broadcast, the sender first computes a *cover* of the current members. A cover is a choice of subsets from the family, so that all members belong to at least one chosen subset, and no revoked user belong to any chosen subset. The message broadcasted will then contain, for each subset in the cover, the message key encrypted under that subset's key.

Without key derivation, each user would have to store the key for each subset of which she is a member. However, when using key derivation, keys of subsets are related in a way that allows a user to derive keys of subsets by applying a suitable function, typically a one way function, to her set of secrets. Thus the space decreases, as one secret can be used to derive multiple keys.

Example 1. Figure 1(a) shows an example of a Subset Cover scheme on $n = 4$ users. In the example, the family of subsets consists of all four singleton subsets, four subsets of size 2, and two subsets of size 3. An edge from S_i to S_j indicates that the secrets used to derive the key for S_i can also be used to derive the key for S_j. Thus, the secret used by user 2 to derive the key for her singleton set $\{2\}$ can also be used to derive the keys for nodes $\{1,2\}$, $\{2,3\}$, and $\{1,2,3\}$. Without key derivation, she would have had to store four keys, but now she only needs to store one secret.

More formally, a Subset Cover scheme consists of a family of subsets $\mathcal{F} = \{S\} \subseteq 2^{[n]}$ with the property that for every selection of members $M \subseteq [n]$ there is a cover $T \subseteq \mathcal{F}$ such that $\cup_{S \in T} S = M$. There is a set of "secrets" $\mathcal{K}$, and each user $i \in [n]$ is given a subset $P(i)$ of these secrets. Additionally, there is, for each $S \in \mathcal{F}$, a set $K(S) \subseteq \mathcal{K}$ of secrets and a secret key $k(S)$, with the following properties:

– Any user with access to a secret in $K(S)$ can compute $k(S)$.
– For every $S \in \mathcal{F}$ and user $i \in [n]$, $P(i) \cap K(S) \neq \emptyset$ if and only if $i \in S$.
– An adversary with access to all secrets in $\mathcal{K} \backslash K(S)$ cannot compute $k(S)$

To send a message key to the set $M \subseteq [n]$ of members, the cover $T \subseteq \mathcal{F}$ of subsets is chosen in such a way that $\cup_{S \in T} S = M$. The server then broadcasts the message key encrypted using $k(S)$ for each $S \in T$. The bandwidth required for this is $|T|$. We remark that a Subset Cover scheme is required to be able to cover any member set $M \subseteq [n]$.

Naturally, a Subset Cover scheme should also include efficient ways of computing $k(S)$ and the cover T, but as we are interested in lower bounds on the tradeoff between space and bandwidth, these computational issues are not relevant to us.

We denote by $B(\mathcal{F})$ the partially ordered set on the elements of $\mathcal{F}$ in which $S_1 \leq S_2$ if $K(S_1) \subseteq K(S_2)$, i.e., if any secret that can be used to deduce $k(S_1)$ can also be used to deduce $k(S_2)$. Note that $S_1 \leq S_2$ implies $S_1 \subseteq S_2$ (since any user $u \in S_1$ will be able to compute $k(S_2)$ and thus has to be an element of S_2). From now on, we will ignore the set of secrets and the keys, and only study the poset $B(\mathcal{F})$, since it captures all information that we need for our lower bounds. In Figure 1 we show Hasse diagrams of $B(\mathcal{F})$ for two toy example Subset Cover schemes.

The number of secrets a user u needs to store, i.e., the space s, is precisely the number of elements S of $B(\mathcal{F})$ such that u occurs in S, but not in any of the predecessors of S.

Lemma 1. *Any Subset Cover scheme will have at least one singleton node for each user.*

Proof. If there is a user which does not occur in a singleton node, the Broadcast Encryption scheme would fail when the sender attempts to broadcast only to that user. □

2.4 Key Derivation Based on a PRSG

The most common type of key derivation uses a Pseudo-Random Sequence Generator (PRSG), or equivalently, a family of hash functions. This type of key derivation was first used in the context of Broadcast Encryption in the Subset Difference scheme [16]. In [4] it is called Sequential Key Derivation Pattern. The key derivation described here is the intuitive way to do key derivation using a PRSG, and all Subset Cover schemes that the authors are aware of that use a PRSG (or a family of hash functions) do have this form of key derivation.

Let ℓ be a security parameter and let $H(x)$ be a pseudo-random sequence generator taking as seed a string x of length ℓ. Let $H_0(x)$ denote the first ℓ bits of output when running $H(x)$, let $H_1(x)$ denote the next ℓ bits, and so on.

Each subset S in the scheme will be assigned a seed $p(S)$ and a key $k(S)$. The key $k(S)$ will be computed as $k(S) = H_0(p(S))$, so from the seed for a subset, one can always compute the key for that subset. All secrets given to users will be seeds, no user is ever given a key directly. The reason for this is that it gives an almost immediate proof of the security of the scheme by giving the keys the property of *key indistinguishability*, which was proved in [16] to be sufficient for the scheme to be secure in a model also defined in [16].

Consider an edge $e = (S_i, S_j)$ in the Hasse diagram of $B(\mathcal{F})$. The edge means that someone with access to the secrets to deduce $k(S_i)$, i.e. $p(S_i)$ should also be able to deduce $p(S_j)$. If we let $p(S_j) = H_c(p(S_i))$ for some $c \geq 1$, anyone with $p(S_i)$ can derive $p(S_j)$. For a node S_i with edges to $S_{j_1}, S_{j_2}, \ldots, S_{j_k}$ we let $p(S_{j_1}) = H_1(p(S_i)), p(S_{j_2}) = H_2(p(S_i)), \ldots p(S_{j_k}) = H_k(p(S_i))$.

This construction cannot support nodes with indegree greater than 1, since that would require $p(S_j) = H_{c_1}(p(S_{i_1})) = H_{c_2}(p(S_{i_2}))$, which, in general, we cannot hope to achieve. This means that the Hasse diagram will be a forest, since all nodes have an indegree of either 0 or 1.

2.5 UP-Schemes

When the Hasse diagram of $B(\mathcal{F})$ is a forest, we say that the Subset Cover scheme is a *Unique Predecessor scheme* (UP-scheme). Schemes using key derivation as described in Subsection 2.4 will always be UP-schemes. Schemes not using any key derivation (there are no edges in $B(\mathcal{F})$) are also UP-schemes, this class of schemes is sometimes referred to as information-theoretic.

Example 2. The scheme in Figure 1(a) is not a UP-scheme, since there are several sets which have multiple incoming edges, for instance the set $\{1, 2\}$. However, the scheme $\mathcal{S}$ in Figure 1(b) is a UP-scheme. In this case, user 1 would have to store two secrets, one for her singleton node, and one for the node $\{1, 2, 3, 4\}$. The keys for nodes $\{1, 2\}$ and $\{1, 4\}$ can be derived from the same secret used to derive the key for her singleton node.

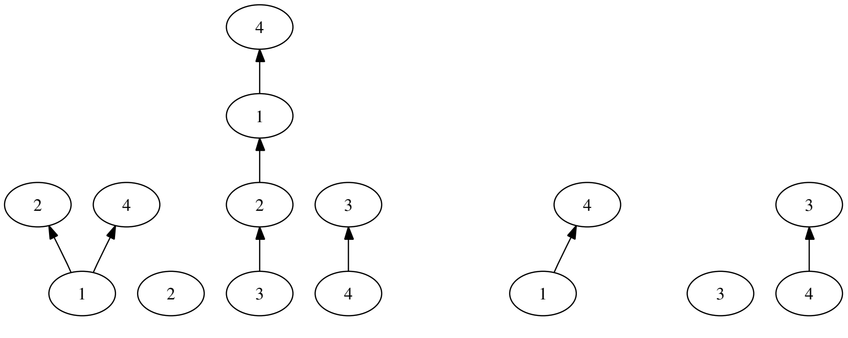
We view a UP-scheme as a rooted forest $\mathcal{S}$, in which each node $V \subseteq [n]$ is labelled with the set of users which are in V, but are not in the parent node. The number of node labels in which a user occurs is the same as the number of secrets that a user will need to store. Thus, when we say that a scheme $\mathcal{S}$ has space s we mean that every user can be used in a label at most s times.

Lemma 2. *Any Unique Predecessor scheme will have at most ns distinct subsets.*

Proof. Adding a new node to a Unique Predecessor scheme means increasing the space for at least one member. Starting from an "empty" scheme, this can be done at most ns times. □

2.6 Normalized UP-Schemes

To simplify the proofs, we will work with *normalized* UP-schemes. We will show that we can perform a simple normalization of a UP-scheme which gives a new scheme with the same set of users, no more space, and at most the same bandwidth. This normalization is similar to the construction of the Flexible SD scheme in [4].



(a) Normalization $\mathcal{S}'$ of the UP-scheme $\mathcal{S}$ in Figure 1(b)

(b) The subscheme $\mathcal{S}'(\{1, 3, 4\})$

Fig. 2. Normalization of UP-schemes

Definition 1. *A UP-scheme $\mathcal{S}$ is* normalized *when every node of $\mathcal{S}$ is labelled with exactly one user and $\mathcal{S}$ has exactly n trees.*

Example 3. The scheme $\mathcal{S}$ from Figure 1(b) is a UP-scheme, but it is not normalized. Two nodes violate the normalization criteria. First, the key for $\{1, 2, 3, 4\}$ can be directly derived from the secrets used for $\{2, 3\}$, which adds two new users at the same time. Second, the node $\{2, 3\}$ also adds two users at once. In Figure 2(a) shows a normalized scheme $\mathcal{S}'$ which is essentially equivalent with $\mathcal{S}$. The key for $\{2, 3\}$ can now be derived from the secret for $\{3\}$, and an extra node $\{1, 2, 3\}$ was inserted between $\{2, 3\}$ and $\{1, 2, 3, 4\}$.

Lemma 3. *Let $\mathcal{S}$ be an arbitrary UP-scheme (on n users) with space s and bandwidth b. Then there exists a normalized UP-scheme $\mathcal{S}'$ (on n users) with space $s' \leq s$ and bandwidth $b' \leq b$.*

Proof. The proof consists of two steps. First we ensure that each node is labelled with exactly one user. Second, we merge identical nodes, which will ensure that $\mathcal{S}'$ has exactly n trees.

Consider a node labelled with a set $U = \{u_1, \ldots, u_k\}$ of users with $k > 1$. Now, split this node into a chain of k nodes, adding one user at a time (in arbitrary order) rather than all k at once. Call the resulting forest $\mathcal{S}_0$. Note that, strictly speaking, it is possible that $\mathcal{S}_0$ is not a UP-scheme as we have defined it, since there may be several nodes representing the same subset S of users. However, it still makes sense to speak of the space and bandwidth of $\mathcal{S}_0$, and we note that the space of $\mathcal{S}_0$ is the same as that of $\mathcal{S}$, as each user occurs in the same number of labels in both. Furthermore, the bandwidth of $\mathcal{S}_0$ is no more than that of $\mathcal{S}$, since all subsets of users present in $\mathcal{S}$ are also present in $\mathcal{S}_0$, and thus, any cover in $\mathcal{S}$ is also valid in $\mathcal{S}_0$.

Next, we describe how to merge nodes representing the same set $S \subseteq [n]$. Given two such nodes v_1 and v_2, attach the children of v_2 as children to v_1, and remove v_2 from the scheme. Note that this operation does not change the bandwidth of the scheme, since only a single node is removed, and this node represents a subset which is still present in the resulting scheme. Also, the space for the resulting scheme will be no larger than the space for the original scheme. The user which was the label for v_2 will now need to store one secret less, whereas the space will be the same for all other users. Let $\mathcal{S}'$ be the result of applying this merging until every set is represented by at most one node.

It remains to show that $\mathcal{S}'$ has exactly n trees. By Lemma 1, there must be at least n trees in $\mathcal{S}'$. Because of the first step every root of a tree will represent a singleton set, and because of the second step every singleton set can be present at most once, implying that there are at most n trees. $\square$

We will, without loss of generality, from now on assume UP-schemes we deal with are normalized. See Figure 2(a) for an example of a normalized UP-scheme. We would like to remark that while normalization can only improve bandwidth and space, it does so at the cost of time. Thus, when applied to improve the performance of practical schemes, one has to take into account the computation time of users, as discussed in [4].

We remark that, in general, normalization will introduce key derivation, even if the original UP-scheme had completely independent keys.

Definition 2. *Given a UP-scheme $\mathcal{S}$ and a set $X \subseteq [n]$ of users, the* subscheme of $\mathcal{S}$ induced by X*, denoted $\mathcal{S}(X)$, is defined as follows: for every user $y \notin X$, we remove all nodes of $\mathcal{S}$ labelled with y, and their subtrees.*

In other words, $\mathcal{S}(X)$ contains the nodes (and thus subsets) which are still usable when $[n] \setminus X$ have been revoked. See Figure 2(b) for an example.

3 Few Revoked Users

We prove that when the number of revoked users r is small, any UP-scheme using at most polynomial space will require bandwidth $\Omega(r)$.

As noted in the introduction, the requirement that the space is polynomial is *very* generous. Anything beyond polylogarithmic space per user is generally considered impractical.

Theorem 1. *Let $c \geq 0$ and $0 \leq \delta < 1$. Then, any UP-scheme with n users and space $s \leq n^c$ will, when the number of revoked users $r \leq n^\delta$, require bandwidth*

$$b \geq \frac{1-\delta}{c+1} \cdot r \tag{1}$$

Proof. Let $\mathcal{S}$ be an arbitrary UP-scheme with $s \leq n^c$ and let $r \leq n^\delta$. An upper bound on the number t of sets of users that can be handled using bandwidth at most b is given by the number of sets of nodes of $\mathcal{S}$ of cardinality at most b. Since $\mathcal{S}$ contains at most ns nodes, this is upper-bounded by

$$\sum_{i=1}^{b} \binom{ns}{i} \leq (ns)^b \leq n^{(c+1)b} \tag{2}$$

In order for $\mathcal{S}$ to be able to handle every set of revoked users of size r, we need t to be at least $\binom{n}{r}$, giving

$$n^{(c+1)b} \geq \binom{n}{r} \geq (n/r)^r \geq n^{(1-\delta)r} \tag{3}$$

and the theorem follows. $\square$

Theorem 1 comes very close to matching many of the previous works, for instance Subset Difference [16] with $s = \mathcal{O}(\log^2 n)$ and $b = \min(2r-1, n-r)$. For $r \leq \sqrt{n}$, our bound gives $b \geq \frac{r}{2(1+c)}$ which is within a factor $4 + o(1)$.

As mentioned in the introduction, [9] has shown a stronger bound, roughly $r\frac{\log(n/r)}{\log(rs)}$, using the Sunflower lemma. However, their bound applies only to Subset Cover schemes without key derivation, and is in fact stronger than existing schemes using key derivation – e.g. the Subset Difference scheme mentioned above for $r < n^{1/3}$.

4 Arbitrarily Many Revoked Users

In this section we show that, for a certain choice of r, any UP-scheme has to use bandwidth at least $\frac{n}{1.89 \log s}$. We start with Theorem 2, which gives a lower bound on the bandwidth as a function of m/n. Plugging in a suitable value of m/n in Corollary 1 will then give the desired result.

Theorem 2. *Let $\delta \in (0, 1]$ and $\epsilon > 0$. Then for every UP-scheme $\mathcal{S}$ with $n > \frac{2\delta(1-\delta)}{\epsilon^2}$ there exists a set of users M of size $\delta - 3\epsilon \leq |M|/n \leq \delta + \epsilon$ which requires bandwidth $b \geq |M| \frac{\log(1/\delta)}{\log(s/\epsilon)}$*

Proof. Pick $M_0 \subseteq [n]$ randomly where every element is chosen with probability δ, independently.

Set $d = \log_\delta(\epsilon/s)$ and let X be the set of users which occur at depth exactly d in $\mathcal{S}(M_0)$ (where the roots are considered to be at depth 1). Let $M = M_0 \setminus X$. Since each node can cover at most d users of M, the bandwidth required for M is at least

$$\frac{|M|}{d} = |M| \frac{\log(1/\delta)}{\log(s/\epsilon)}$$

It remains to show that there is a positive probability (over the random choice of M_0) that M ends up having the required size, as this implies that such an M exists.

The probability that a node at depth d of $\mathcal{S}$ remains in $\mathcal{S}(M_0)$ is $\delta^d = \epsilon/s$. The total number of nodes at depth d in $\mathcal{S}$ is upper-bounded by ns, and thus, the expected number of nodes at depth d in $\mathcal{S}(M_0)$, i.e. the expected size of X, is at most $\delta^d ns = \epsilon n$. By Chebyshev's inequality, we have $\Pr\left[\left|\frac{|M_0|}{n} - \delta\right| \geq \epsilon\right] \leq \frac{\delta(1-\delta)}{n\epsilon^2} < 1/2$. By Markov's inequality, we have $\Pr\left[\frac{|X|}{n} \geq 2\epsilon\right] \leq 1/2$. The union bound then gives that $\Pr[\delta - 3\epsilon \leq |M|/n \leq \delta + \epsilon] > 0$. Thus, there exists some choice of M_0 such that $|M|$ falls within this range. $\square$

As a corollary, we have:

Corollary 1. *For any $\epsilon > 0$ there exist n_0 and s_0 such that any UP-scheme $\mathcal{S}$ with $n \geq n_0$ and $s \geq s_0$ uses bandwidth at least*

$$\frac{n}{(e \ln(2) + \epsilon) \log_2(s)} \approx \frac{n}{1.89 \log_2(s)} \tag{4}$$

Proof. Let $\delta = 1/e$. Invoking Theorem 2 with parameters δ and ϵ' (the value of which will be addressed momentarily), we get a set M of size at least $(\delta - 3\epsilon')n$ requiring bandwidth at least

$$n \frac{\delta - 3\epsilon'}{\ln(s/\epsilon')} = n \frac{1 - 3e\epsilon'}{e \ln(2) \log_2(s) + e \ln(1/\epsilon')} \tag{5}$$

Pick ϵ' small enough so that

$$\frac{e \ln(2)}{1 - 3e\epsilon'} \leq e \ln(2) + \epsilon/2.$$

Then Equation (5) is lower-bounded by Equation (4) for any s satisfying

$$\frac{e\ln(1/\epsilon')}{1-3e\epsilon'} \le \frac{\epsilon}{2}\log_2(s)$$
$$\log_2(s) \ge \frac{2e\ln(1/\epsilon')}{\epsilon(1-3e\epsilon')},$$

and we are done. □

We remark that Corollary 1 is tight up to the small constant 1.89, as seen by the following theorem.

Theorem 3. *There exists a UP-scheme $\mathcal{S}$ using bandwidth at most $\left\lceil \frac{n}{\log_2(s)} \right\rceil$.*

Proof. Partition the users into $\lceil n/\log_2(s) \rceil$ blocks of size $\le \log_2(s)$. Then, in each block, use the naive scheme with exponential space and bandwidth 1, independently of the other blocks. □

5 Bandwidth is $n - r$ for Large r

We show that when the number of revoked users gets very large, all UP-schemes will have a bandwith of $n-r$, e.g. one encryption per member. Exactly how large r has to be for this bound to apply depends on s. This is the same bandwidth as is achieved by the naive solution of just giving each user her own private key.

Theorem 4. *For any UP-scheme $\mathcal{S}$ and $m \le \frac{n}{6s}$, there is a member set M of size $|M| = m$ requiring bandwidth $b = |M|$.*

Proof. We will build a sequence $M_0 \subseteq M_1 \subseteq M_2 \subseteq \ldots \subseteq M_m$ of sets of members with the properties that $|M_i| = i$, and that the bandwidth required for M_i is i.

The initial set M_0 is the empty set. To construct M_{i+1} from M_i, we pick a user $u \notin M_i$ satisfying:

- There is no $v \in M_i$ such that some node labelled with u occurs as the parent of some node labelled with v
- There is no $v \in M_i$ such that the root node labelled with v occurs as the parent of some node labelled with u
- The root node labelled u has outdegree $\le 2s$

We then set $M_{i+1} = M_i \cup \{u\}$. Clearly $|M_i| = i$, so there are two claims which remain to be proved. First, that the required bandwidth of M_i is i. Second, that the process can be repeated at least m times.

To compute the bandwidth of M_i, we prove that the only way to cover M_i is to pick the singleton sets of every $u \in M_i$. To see this, assume for contradiction that there exists some set S with $|S| > 1$ that can be used when constructing a cover. This corresponds to a node x at depth $|S|$ of some tree, and S is given by the labels of all nodes from x up to the root. In order for us to be able to use

S when constructing a cover, all these nodes need to belong to M_i. However, the first two criterions in the selection of u above guarantee that not all of these nodes can belong to M_i. The first criterion states that, once we have added a node, we can never add its parent. The second criterion states that, once we have added a root, we can never add any of its children. This shows that there can be no such S.

To see how many steps the process can be repeated, let r_i be the total number of nodes which are "disqualified" after having constructed M_i. Then, M_{i+1} can be constructed if and only if $r_i < n$. First, r_0 equals the number of roots which have degree $> 2s$. Since the total number of nodes is at most ns, this number is at most $r_0 \leq n/2$. When going from M_i to M_{i+1}, the total number of new disqualified nodes can be at most $3s$ – the node added, the parents of the at most $s-1$ non-root occurrences of u, and the at most $2s$ children of the root labelled with u. Thus, we have that $r_i \leq n/2 + 3si$, which is less then n if $i < \frac{n}{6s}$. □

The lower bound of Theorem 4 is tight up to a small constant in the following sense.

Theorem 5. *There exists a UP-scheme $\mathcal{S}$ such that for any set M of $|M| > \lceil \frac{n}{s} \rceil$ members, the bandwidth is $b < |M|$.*

Proof. Partition the set of users into $B = \lceil \frac{n}{s} \rceil$ blocks of size $\leq s$, and let each user share a key with each of the $s-1$ other users in her block. Then, given a set M of size $|M| > B$, there must be two users $i, j \in M$ belonging to the same block. Using the key shared by i and j to cover both them both, we see that the bandwidth of M is at most $b \leq |M| - 1$. □

6 Conclusion

In this paper, we have shown lower bounds for a large class of Subset Cover based Broadcast Encryption schemes. This type of scheme is probably the most explored class of schemes today, with many constructions. Our proofs are in a model with very relaxed constraints compared to what is considered practical, so it would not help to simply relax requirements slightly (e.g. allowing more space or time). The lower bounds shown in this paper match known constructions very well.

In particular, our bounds show that it will be impossible to get a bandwidth of $o(r)$ without increasing the space requirements to unreasonable levels or using some new form of key derivation. We do not have any lower bounds on the memory needed for $\mathcal{O}(r)$ bandwidth, an open question is thus if it is possible to get $\mathcal{O}(r)$ bandwidth with space $o(\log n)$.

Acknowledgements. The authors are grateful to Johan Håstad for many useful comments and discussions.

References

1. Adelsbach, A., Greveler, U.: A broadcast encryption scheme with free-riders but unconditional security. In: Safavi-Naini, R., Yung, M. (eds.) DRMTICS 2005. LNCS, vol. 3919, pp. 246–257. Springer, Heidelberg (2006)
2. Asano, T.: A revocation scheme with minimal storage at receivers. In: Zheng, Y. (ed.) ASIACRYPT 2002. LNCS, vol. 2501, pp. 433–450. Springer, Heidelberg (2002)
3. Asano, T.: Reducing Storage at Receivers in SD and LSD Broadcast Encryption Schemes. In: Chae, K.-J., Yung, M. (eds.) WISA 2003. LNCS, vol. 2908, pp. 317–332. Springer, Heidelberg (2004)
4. Attrapadung, N., Kobara, K., Imai, H.: Sequential key derivation patterns for broadcast encryption and key predistribution schemes. In: Laih, C.S. (ed.) ASIACRYPT 2003. LNCS, vol. 2894, pp. 374–391. Springer, Heidelberg (2003)
5. Berkovits, S.: How to broadcast a secret. In: Davies, D.W. (ed.) EUROCRYPT 1991. LNCS, vol. 547, pp. 535–541. Springer, Heidelberg (1991)
6. Boneh, D., Gentry, C., Waters, B.: Collusion resistant broadcast encryption with short ciphertexts and private keys. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 258–275. Springer, Heidelberg (2005)
7. Fiat, A., Naor, M.: Broadcast encryption. In: Stinson, D.R. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 480–491. Springer, Heidelberg (1994)
8. Gentry, C., Ramzan, Z.: RSA accumulator based broadcast encryption. In: Zhang, K., Zheng, Y. (eds.) ISC 2004. LNCS, vol. 3225, pp. 73–86. Springer, Heidelberg (2004)
9. Gentry, C., Ramzan, Z., Woodruff, D.P.: Explicit exclusive set systems with applications to broadcast encryption. In: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), pp. 27–38. IEEE Computer Society, Washington (2006)
10. Goodrich, M.T., Sun, J.Z., Tamassia, R.: Efficient tree-based revocation in groups of low-state devices. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 511–527. Springer, Heidelberg (2004)
11. Halevy, D., Shamir, A.: The LSD broadcast encryption scheme. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 47–60. Springer, Heidelberg (2002)
12. Hwang, J.Y., Lee, D.H., Lim, J.: Generic transformation for scalable broadcast encryption schemes. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 276–292. Springer, Heidelberg (2005)
13. Jho, N.S., Hwang, J.Y., Cheon, J.H., Kim, M.H., Lee, D.H., Yoo, E.S.: One-way chain based broadcast encryption schemes. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 559–574. Springer, Heidelberg (2005)
14. Johansson, M., Kreitz, G., Lindholm, F.: Stateful subset cover. In: Zhou, J., Yung, M., Bao, F. (eds.) ACNS 2006. LNCS, vol. 3989, pp. 178–193. Springer, Heidelberg (2006)
15. Luby, M., Staddon, J.: Combinatorial bounds for broadcast encryption. In: Nyberg, K. (ed.) EUROCRYPT 1998. LNCS, vol. 1403, pp. 512–526. Springer, Heidelberg (1998)
16. Naor, D., Naor, M., Lotspiech, J.: Revocation and tracing schemes for stateless receivers. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 41–62. Springer, Heidelberg (2001)

VII

Stateful Subset Cover

Mattias Johansson[1], Gunnar Kreitz[2], and Fredrik Lindholm[1]

[1] Ericsson AB, SE-16480 Stockholm, Sweden
{mattias.a.johansson, fredrik.lindholm}@ericsson.com
[2] Royal Institute of Technology, Stockholm, Sweden
gkreitz@nada.kth.se

Abstract. This paper describes a method to convert stateless key revocation schemes based on the subset cover principle into stateful schemes. The main motivation is to reduce the bandwidth overhead to make broadcast encryption schemes more practical in network environments with limited bandwidth resources, such as cellular networks. This modification is not fully collusion-resistant.

A concrete new scheme based on the Subset Difference scheme [1] is presented, accomplishing a bandwidth overhead of $\Delta m + 2\Delta r + 1$ compared to e.g. Logical Key Hierarchy's $2(\Delta m + \Delta r)\log m$, where Δm and Δr is the number of members added and removed since the last stateful update and m is the number of current members.

Keywords: Broadcast encryption, key revocation, subset cover, Subset Difference, Logical Key Hierarchy, stateful, stateless.

1 Introduction

In this paper we show how a key server can establish a common group key K_g for a dynamically changing group (i.e., members can join and leave). One possible application area is the protection of broadcast streams (e.g., internet or mobile broadcasting of movies, music, or news), and the topic is therefore generally referred to as broadcast encryption. The group key which is to be distributed is often referred to as the media key, or session key.

This problem is well studied and is usually solved by using a key revocation scheme. One large class of key revocation schemes are the subset cover schemes, introduced in [1]. In this paper we present a general method for adding state to subset cover schemes, which reduces the bandwidth overhead greatly.

In the system setup stage, the key server gives each user u some key information K_u. This information can be thought of as a set of keys; in general it will be information from which keys can be derived. The size of K_u is called the *user storage*. Schemes where K_u is never updated are called *stateless*, whereas those where it is updated K_u are called *stateful*.

Every time a new group key is distributed, the key server will broadcast a *header*, using which all legitimate group members can calculate the new group key. The size of this header is called the *bandwidth overhead*, and the time it takes for a member to compute the group key from the header and her set of key information K_u is called the *computational overhead*.

1.1 Preliminaries

Broadcast encryption was first introduced by Berkovits in [2], and later Fiat and Naor started a more formal study of the subject [3]. The first practical broadcast encryption scheme was the stateful Logical Key Hierarchy (LKH) scheme proposed in [4, 5]. LKH accomplishes a worst case bandwidth overhead of $2(\Delta m + \Delta r)\log m$, where Δm and Δr are the number of added and removed members since the last stateful key update, and m is the number of current members.

Later, the class of schemes known as subset cover schemes, and the Subset Difference (SD) scheme were presented in [1]. Further variants of the SD scheme have been developed in [6, 7]. Other subset cover schemes include the Hierarchical Key Tree scheme [8], and the Punctured Interval scheme, π [9]. All of these schemes have bandwidth overhead which is linear in r.

Stateful or Stateless. The advantage of a stateless scheme compared to a stateful scheme is that a member does not have to receive all previous updates in order to decrypt the current broadcast. In many settings, this advantage is not as big as it first appears. Stateful schemes can be augmented with reliable multicast techniques or can make missed broadcasts available on request. Also, in e.g. a commercial settings where the group key is updated every five minutes, a stateless scheme would also need to use similar techniques, since missing five minutes of content due to a single packet lost would be unacceptable.

Notation. Let $\mathcal{M}$ be the set of members of the group, $\mathcal{R}$ be the set of revoked users and $\mathcal{U}$ be the total set of users, or potential members (i.e. the union of $\mathcal{M}$ and $\mathcal{R}$). Let m, r and u be the sizes of these sets. Let Δm and Δr be the number of users who have joined and left the group since the last *stateful* update. Let $E_K(M)$ be the encryption of message M under key K. In a binary tree, let $\mathrm{l}(v)$ and $\mathrm{r}(v)$ be the left and right child of node v. Let $\mathrm{par}(v)$ and $\mathrm{sib}(v)$ be the parent and sibling of node v.

1.2 Our Contribution

Subset cover schemes define a family of subsets of $\mathcal{U}$, where each subset is associated with a key. To distribute a new group key, the key server covers $\mathcal{M}$ (and avoids $\mathcal{R}$) with subsets from the family and encrypts the new group key K_g using the key of each subset used in the cover. We present a technique where a *state key*, K_S is added, which is held by current members of the group.

When distributing a new group key, the state key is used to transform all subset keys. Since only current members have access to the state key, the key server does not need to avoid covering all of $\mathcal{R}$, but only those who were recently removed (and thus have a current state key).

This technique can be applied to any scheme based on the subset cover principle. It is often beneficial to develop a new algorithm to calculate the cover, and this has been done for the SD scheme.

1.3 Organization of This Paper

In Section 2, a brief overview of subset cover schemes is given. In Section 3, our idea, Stateful Subset Cover, is presented in detail. In Section 4, we show practical performance results on some simulated datasets. In Section 5, we discuss the security of our proposed scheme. We give concluding remarks in Section 6. Algorithms for Stateful Subset Cover are given in Appendix A.

2 Subset Cover Schemes

A general class of stateless schemes are called subset cover schemes and were first introduced in [1]. In this class of revocation schemes, there is a preconfigured family of sets, $\mathcal{F} = \{f_1, f_2, \ldots\}, f_i \subseteq \mathcal{U}$. Each set $f_i \in \mathcal{F}$ has an associated key K_i such that each user belonging to f_i can compute K_i, but no user outside of f_i can compute K_i.

To distribute a new group key, the key server calculates an exact cover $\mathcal{F}'$ of $\mathcal{M}$, i.e. $\mathcal{F}' = \{f_{i_1}, f_{i_2}, \ldots\} \subseteq \mathcal{F}$ and $\bigcup \mathcal{F}' = f_{i_1} \cup f_{i_2} \cup \ldots = \mathcal{M}$. The key server then broadcasts the following message:

$$\mathcal{F}', \mathrm{E}_{K_{i_1}}(K_g), \mathrm{E}_{K_{i_2}}(K_g), \ldots$$

where $\mathcal{F}'$ here denotes some suitable representation of $\mathcal{F}'$ such that members can compute what part of the message to decrypt using what key. Since the sets $\mathcal{F}$ and the keys associated with the sets are fixed, this broadcast encryption scheme is stateless.

2.1 Subset Difference

In the Subset Difference (SD) scheme, which is a subset cover scheme, every user is associated with a leaf in a binary tree. For every node v in the tree, and every node w below v, we have $S_{v,w} \in \mathcal{F}$, where $S_{v,w}$ is the set of all leaves in the subtree rooted in v, except for those in the subtree rooted in w. In figure 1, two such sets are shown, the set $S_{2,10}$ and the set $S_{6,12}$. The corresponding broadcast from the key server would in this case be

$$\{S_{2,10}, S_{6,12}\}, E_{K_{S_{2,10}}}(K_g), E_{K_{S_{6,12}}}(K_g).$$

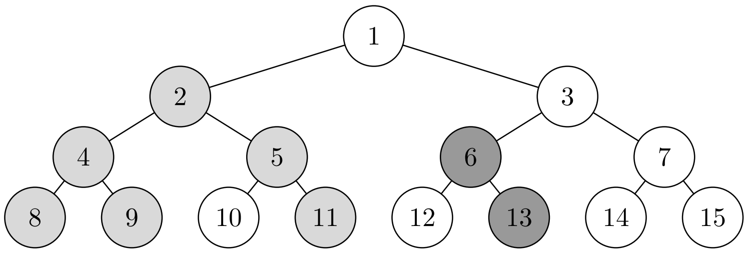


Fig. 1. The sets $S_{2,10}$ (light) and $S_{6,12}$ (dark) in an SD tree

A user is not given the keys $K_{S_{v,w}}$ she is entitled to directly, since that would consume too much user memory. Instead, she is given $\mathcal{O}(\log^2 u)$ values from which the keys she should have access to can be derived by $\mathcal{O}(\log u)$ applications of a pseudo-random number generator. For details on how the key derivation in SD works, see [1]. The SD scheme has a bandwidth overhead which is $\min(2r+1, m)$.

2.2 The Punctured Interval Scheme π

In the punctured interval (π) scheme, users can be thought of as being on a line, each user indexed by an integer. The subsets used in the π scheme are of the form $S_{i,j;x_1,\ldots,x_q} = \{x|i \leq x \leq j, x \neq x_k, 1 \leq k \leq q\}$, i.e. all users between positions i and j (inclusive) except for the q users $x_1, \ldots, x_q$.

The scheme has two parameters, p and c affecting the performance of the system. The parameter c is the maximum length of the interval, e.g. $1 \leq j - i + 1 \leq c$, and the parameter p is how many users in the interval can at most be excluded, e.g. $0 \leq q \leq p$. Large p and c lower bandwidth requirements but increase user storage and computational overhead. The bandwidth overhead is about $\frac{r}{p+1} + \frac{u-r}{c}$ and the user storage is $\mathcal{O}(c^{p+1})$. For details on the π scheme, see [9].

3 Stateful Subset Cover

In this section, a general technique for transforming a subset cover scheme into a stateful scheme through the introduction of a *state key* is presented. This makes the bandwidth performance linear in $\Delta m + \Delta r$ instead of in r. As will be discussed further in Section 5, this weakens the security of the system somewhat in that it opens up the opportunity for collaboration. This risk can however be mitigated by periodically using the normal update mechanism of the underlying subset cover scheme, which is referred to as a *hard update.*

3.1 An Intuitive Description

Recall that, as presented in Section 2, a subset cover scheme by covering members using a static family of subsets of users. The subsets are created at startup-time and are constant throughout the life of the system. Each subset is associated with a key that only users in that subset have access to. To distribute a new group key, the key server broadcasts the new group key encrypted with the key for each subset in the cover.

We introduce a general extension to a subset cover scheme by adding a state key, which is distributed alongside with the group key to members. This state key is then used for distributing the next group key and state key. When the key server broadcasts a new group key (and state key) it will not encrypt the new group key directly with the keys of the selected subsets, instead it will use the key of the selected subset transformed by the current state key using some suitable function (e.g. xor). This means that to decrypt the new group key, a user must not only be in a selected subset, but must also have the current state key.

The key server, when it saves bandwidth doing so, is thus free to cover revoked users too, as long as they do not have access to the current state key. So, the key server need only avoid covering those who were revoked recently. To discourage cheating (see Section 5) the aim is to cover as few revoked users as possible, which is referred to as a *cheap* cover.

The alert reader may have noticed a problem with the system as described. If the state key is needed to decrypt the new group key and state key, how are recently joined members, who do not have the current state key, handled? The answer is that the state key is not used when covering joiners, and thus all of $\mathcal{R}$ must be avoided. However, the scheme is free to cover current members who have a state key, and it is preferable for it to cover as many of these as it can, since those covered here will not need to be covered in the cover using the state key. This is called a *generous* cover.

A variation of the above extension is to run the system in a *semi-stateless* mode. That means that the key server in each round is free to decide whether it wishes to update the state key or not. As long as the state key is not updated, the scheme will have the properties of a stateless scheme, but the bandwidth usage will gradually increase since Δm and Δr (membership changes since last *stateful* update, see Section 1.1) will increase. A group key update when the state key is changed is called a *stateful update* and one where the state key remains unchanged is called *stateless update.*

3.2 Generalized Stateful Subset Cover

For this type of scheme to work, a new cover function is needed. The traditional subset cover has two types of users: members and revoked, or blue and red. The new cover function has three types of users: must cover (MC), can cover (CC) and must not cover (NC). The output is a cover covering all users marked MC and not covering any NC users. Users marked as CC, can be either covered or not covered.

As discussed in Section 3.1, there are two versions of the cover algorithm for each scheme, *generous* and *cheap*. Both versions will primarily minimize the number of subsets used for the cover. The generous cover will attempt to cover as many CC users as possible and the cheap cover will cover as few as possible.

More formally, we have a new decision problem, Optional-Set-Cover($\mathcal{F}$, $\mathcal{M}$, $\mathcal{R}$, k, n), where $\mathcal{F}$ is a family of subsets of some finite set $\mathcal{U}$, $\mathcal{M}$ is subset of the same $\mathcal{U}$ and k and n are integers. The problem is: is there a subset $\mathcal{F}' \subseteq \mathcal{F}$ such that $\bigcup \mathcal{F}' \supseteq \mathcal{M}$, $|\mathcal{F}'| = n$, $|\bigcup \mathcal{F}'| = k$, and $(\bigcup \mathcal{F}') \cap \mathcal{R} = \emptyset$?

The two optimization problems, generous and cheap, both primarily want to minimize n, and then on the second hand either want to maximize or minimize k, respectively. The optimization problems are denoted Generous-Cover($\mathcal{F}$, $\mathcal{M}$, $\mathcal{R}$) and Cheap-Cover($\mathcal{F}$, $\mathcal{M}$, $\mathcal{R}$). The (optional) subset cover problem is in the general case NP complete, but subset cover schemes are designed in such a way that an efficient algorithm exists.

The Framework. The system is initialized exactly as the underlying subset cover scheme, with one exception. A state key, K_S, is generated by the key server and given to all initial members of the system. The key server also keeps track of the set of users to which it has given the current state key, the set $\mathcal{S}$.

To update the group key, the key server first decides whether it is time to do a hard update or not. If a hard update is done, it uses the underlying scheme to distribute a new K'_g and K'_S and sets $\mathcal{S} \leftarrow \mathcal{M}$.

If it was not time to do a hard update, it begins by calculating a cover $\mathcal{C}_1$ as $\mathcal{C}_1 \leftarrow \textsc{Generous-Cover}(\mathcal{F}, \mathcal{M} \backslash \mathcal{S}, \mathcal{R})$. Note that this cover is empty if $\mathcal{M} \backslash \mathcal{S}$ is empty, i.e. if no new members have been added since K_S was last updated.

After this, it checks if $\mathcal{R} \cap \mathcal{S} = \emptyset$. If this is the case, i.e. no one has been removed since K_S was last updated, no one besides members has K_S, and thus K_S can be used to securely communicate with members. If $\mathcal{R} \cap \mathcal{S} \neq \emptyset$, it instead calculates $\mathcal{C}_2 \leftarrow \textsc{Cheap-Cover}(\mathcal{F}, \mathcal{M} \backslash (\bigcup \mathcal{C}_1), \mathcal{R} \cap \mathcal{S})$.

The key server then first broadcasts a description of the covers $\mathcal{C}_1$, $\mathcal{C}_2$ in some form, so that members know what part of the broadcast to decrypt. The message will then consist of, firstly, for every $c \in \mathcal{C}_1$: $\mathrm{E}_{K_c}(K_E)$, where K_c is the key associated with subset c. Secondly, if $\mathcal{R} \cap \mathcal{S} \neq \emptyset$, the message will contain, for every $c \in \mathcal{C}_2$: $\mathrm{E}_{K_c}(K_F)$, or if $\mathcal{R} \cap \mathcal{S} = \emptyset$, $\mathrm{E}_{K_S}(K_F)$. We let $K_E = f(K_F, K_S)$, where f is a suitable function, such as xor. Finally, the message will contain $\mathrm{E}_{K_E}(K'_g, K_S)$ or $\mathrm{E}_{K_E}(K'_g, K'_S)$ depending on if it was a stateless or stateful update, respectively. If the update was stateful, the key server sets $\mathcal{S} \leftarrow \mathcal{M}$, otherwise $\mathcal{S}$ is left unchanged.

Generic Cover. A normal subset cover algorithm can be used to solve the optional subset cover problem, but generally not optimally. Since most subset cover schemes have bandwidth performance which linear in r, it is often beneficial to minimize $\mathcal{R}$.

So, for the optional subset cover problem, we can simply re-mark all CC users as members and then run the normal subset cover algorithm on the resulting set. Post-processing can be done to remove any sets covering only users who were labelled CC before the re-marking. Post-processing can also, in the cheap variant, attempt to narrow a set down (i.e. change the set so that fewer CC users are covered).

For a specific underlying scheme, it is often possible to make better use of the CC users than this rather naïve transformation. An optimal algorithm for the SD scheme will be discussed in the next section.

3.3 Stateful Subset Difference

For SD (Section 2.1), which is one of the most important subset cover schemes, a new cover algorithm has been developed. Pseudo-code for the algorithm can be found in Appendix A, but we describe and discuss it here.

The complexity of the algorithm has the same asymptotic complexity as the original. This algorithm could also be used in stateful variants of other subset cover schemes which use the SD cover method, such as LSD and SSD ([6, 7]).

Let each node v have three variables, two booleans, v**.mc** and v**.nc** and one integer, v**.cc**. The variable v**.cc** counts the number of CC users which can be excluded under v. If v**.mc** is true, it means that there are uncovered MC users (i.e., users which must be covered) below v. If v**.nc** is true, it means that there are NC users (i.e., users which must not be covered) below v.

A basic observation is that a node v where both the left ($\mathrm{l}(v)$) and the right child ($\mathrm{r}(v)$) has a NC node below it cannot be used as a top node for a key. So, if any MC nodes are below such a node, the top node to use for the set cannot be higher up in the tree than $\mathrm{l}(v)$ or $\mathrm{r}(v)$.

For the algorithms given, we assume that the bottom nodes (i.e., the user nodes) have already been colored in the input. That means that for a MC user at node v, v**.mc** = **true**, v**.nc** = **false** and v**.cc** = 0. Analogously for a NC user. For a CC user at node v, v**.mc** = v**.nc** = **false** and v**.cc** = 1.

The algorithm consists of three functions. Cover() is the top-level function which is called to generate the entire cover, with a parameter telling it if a generous or cheap cover is wanted. Cover() "adds up" the marks of the child nodes, and call a helper-function to calculate the exact subset when it discovers that a subset has to be placed. When subsets are placed, they are guaranteed to cover all remaining MC users under the current node. This means that there is nothing left to be covered below that node, so the parent will be marked with only NC, instead of the usual "sum" of the child nodes.

The two functions, Generous-Find-Subset() and Cheap-Find-Subset() calculate a single subset, given a top node which is the highest place for the top node of the subset. Both versions use the markings placed by Cover() to calculate the subset. The generous version will just ensure that no NC users are covered, while the cheap version will both ensure that no NC users are covered and will attempt to exclude as many CC users as possible.

As an example, consider the tree shown in Figure 2. Let the letter 'N' denote v**.nc** = **true**, 'M' denote v**.mc** = **true** and 'C' denote v**.cc** = 1. The bottom nodes have all been colored in the input to the algorithm. Going through the nodes in depth-first (left-to-right) order, in the first node, the NC and CC marks from the child nodes are combined into the parent. In its sibling, the CC and NC marks propagate up.

When we get to the left child of the root, then both children are marked with NC and at least one child is marked with MC. This will cause Find-Subset() to be called. In both cases, Find-Subset() will select the subset $S_{4,8}$, which covers the single MC node 9.

For the right subtree, marks will propagate upwards and the coloring will reach the right child of the root without Find-Subset() being called. However, both children of the root are marked NC, and node 3 is marked MC, so one of the Find-Subset() functions will be called again. In this case, the subset $S_{3,14}$ will be selected, which covers the two MC users 12 and 14 and the CC user 13.

This algorithm differs somewhat from the original cover algorithm for SD given in [1]. The original algorithm begins by calculating the Steiner tree of the revoked users and the root and then calculate the cover directly from the

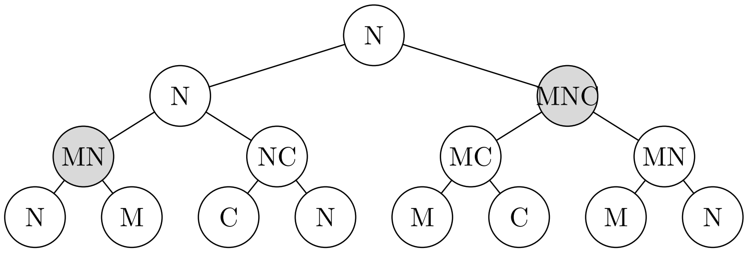


Fig. 2. A sample coloring of a stateful subset difference tree, gray nodes show on what nodes FIND-SUBSET() are called

properties of the Steiner tree. The algorithm presented here also works as a cover algorithm for the normal SD cover problem. Both algorithms have the same time complexity.

3.4 Stateful Punctured Interval

A very simple greedy (and suboptimal) cover algorithm has been tested with the π scheme. We do not present this algorithm here. Recall (Section 2.2) that this scheme has two parameters which can be tuned. Tests have been performed with four sets of parameters, $(c,p) = (1000,1),(100,2),(33,3),(16,4)$. These were selected such that user storage would be approximately 1 Mbyte, which we deemed reasonable for many scenarios. We present the results parameters giving the best results on our dataset, $(c,p) = (1000,1)$. Further tuning may give even better performance.

3.5 Performance

The user storage will essentially be unchanged (a single extra key needs to be stored by members and the key server) by the addition of state, so they will be the same as those of the underlying scheme. Analogously for the computational overhead. The scheme will take on the negative properties of a stateful scheme in that packet loss becomes a more serious issue which will need to be handled, see the discussion in Section 1.1, where we argue that this is not as big a drawback as it first appears.

Bandwidth Impact. The bandwidth performance will in general improve. The bandwidth usage for the first cover calculated (for joining members, where the state key is not used) is at worst that of the underlying scheme with Δm members and r revoked users. For the second cover, where the state key is used, the performance is at worst that of the underlying scheme with m members and Δr revoked users. Inserting the values for SD, we get a worst-case bandwidth performance of $\min(2r+1, \Delta m)+\min(2\Delta r+1, m)$, which will, in most situations, be $\Delta m + 2\Delta r + 1$. This can be compared to for instance LKH, which has a performance of $2(\Delta m + \Delta r) \log m$.

The worst-case performance is better than previous protocols. A comparative performance evaluation in several usage scenarios has also been performed, and some of these results are presented in Section 4.

Computational Complexity for General Stateful Subset Cover. At most u users can be undecided, so at worst, u nodes must be re-labeled. The output of the cover will cover each undecided or cover node exactly once, so the cost of going through the sets is at most the number of such nodes, which is u. Thus, the added runtime for both the pre-processing and post-processing steps is $\mathcal{O}(u)$.

Computational Complexity for Stateful Subset Difference. The performance of the cover algorithm is $\mathcal{O}(u)$. On the way up, each node will be visited exactly once and the tree has $\mathcal{O}(u)$ nodes. When Find-Subset() is called, the top node will only be colored red. All downward traversal in the Find-Subset() functions will always stop at a node colored only red. This means that on the way down (i.e., in Find-Subset()) each node can be visited at most twice.

3.6 Correctness

The framework is correct in that a member can recover the new key, as long as she has not missed the last stateful update. For a member there are two cases. Either, she is recently added and does not have the current state key, or she has the current state key.

If a member m does not have the current state key, then $\mathcal{M} \backslash \mathcal{S} \neq \emptyset$ since it must at least contain m. If so, the cover $\mathcal{C}_1 = \text{Generous-Cover}(\mathcal{M} \backslash \mathcal{S}, \mathcal{R})$ will be calculated and the underlying scheme will be used to distribute the keys with the resulting cover. If the underlying scheme is correct, the member will be able to recover the key and the current state key.

If, on the other hand, m does have the current state key, there are three cases. If m is covered by $\mathcal{C}_1$ then she can discover that fact by looking at the information about the cover and recover the key, given that the underlying scheme was correct.

If $\mathcal{R} \cap \mathcal{S} = \emptyset$, then the new group key will be distributed as $\mathrm{E}_{K_S}(K'_g)$ and since m has K_S, she can recover the new group key.

If $\mathcal{R} \cap \mathcal{S} \neq \emptyset$, then a second cover, $\mathcal{C}_2 = \text{Cheap-Cover}(\mathcal{M} \backslash (\bigcup \mathcal{C}_1), \mathcal{S} \cap \mathcal{R})$ will be calculated. Since m has the state key and was not covered by $\mathcal{C}_1$, she will be covered by $\mathcal{C}_2$. Given that the underlying scheme is correct, she can then recover K_F, from which she can derive K_E since $K_E = f(K_F, K_S)$. She can then recover the new group key which is distributed as $\mathrm{E}_{K_E}(K'_g)$.

4 Practical Results

For simulation purposes, two datasets have been used. In the first dataset, the number of users currently in the group follow a sinus-shaped form, and in the second dataset, the number of users go through (almost) the entire range of the system, from all users being members to almost all users being revoked.

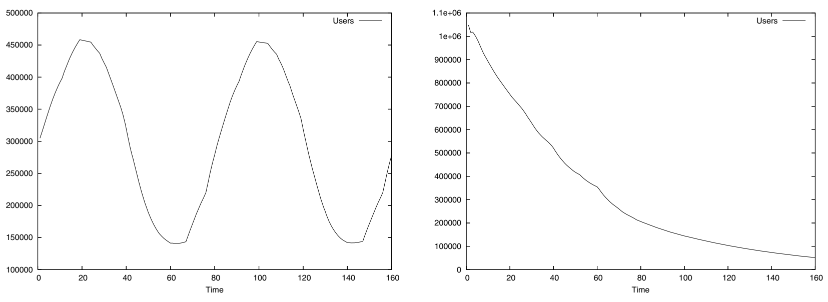


Fig. 3. The sinus-shaped and full-ranged dataset with 2^{20} users

Key updates occur only at discrete intervals (i.e., in batch mode), of which there were 160 in the simulation. To increase the dynamic of the system, except for the joins and leaves necessary to generate the proper form, a *base change* rate was added. The base change is a value between 0 and 1 signifying how large fraction of members that will be replaced by non-members during each round. Given that the scheme we present performs worse in a highly dynamic system, we have used simulations with a very high basechange of 2% to show that it still performs well under difficult conditions. The datasets are displayed in Figure 3. More simulation results are in [10].

The performance of the stateful subset cover schemes presented in this article were evaluated and compared to the performance of the popular LKH scheme, as well as the stateless subset schemes. Note that the stateless schemes do have significantly different performance characteristics, and will e.g. behave poorly when a majority of the population is revoked.

4.1 Performance in Stateful Subset Cover

The performance of the stateful SD and π schemes was evaluated using the scenarios presented in the previous section. As will be shown, the performance is significantly better than that of the LKH scheme, as could be expected from the theoretical analysis.

Figure 4 shows a comparison between two variations of stateful subset cover and LKH. The regular, stateless versions of the subset cover schemes were omitted from this figure for clarity. They do, in fact, have better performance than LKH in this scenario (due to the high base change rate), but the stateful variations still significantly outperform them.

Table 1 shows both the average number of sets used per key update and the maximum number of sets used for a single update. Both these measurements are important to minimize. Minimizing the average will keep total bandwidth usage down and minimizing the maximum will keep the latency for key refresh reasonably low. We show results without any hard updates in these tables. With periodic hard updates, the maximum for the stateful versions will be (about) the same as for the normal versions of the schemes.

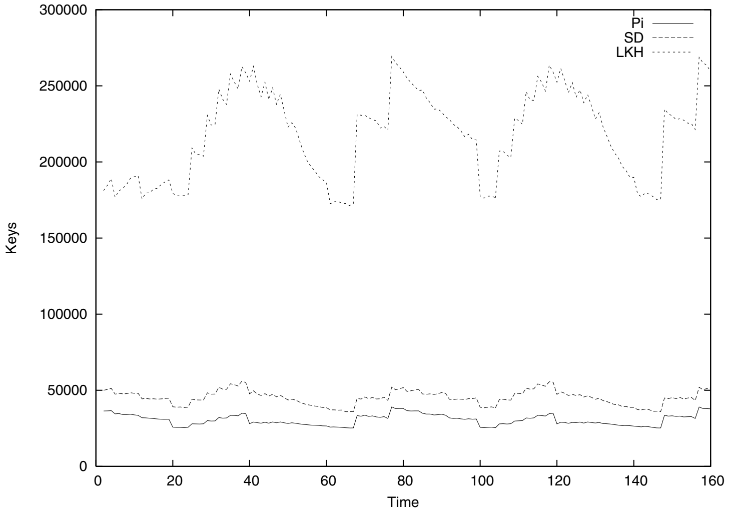


Fig. 4. Bandwidth performance in the sinus-shaped dataset

Table 1. Performance comparison between normal and stateful subset cover schemes (without hard updates), and LKH

	Sinus dataset		Full-range dataset	
Scheme	Avg. sets used	Max sets used	Avg. sets used	Max sets used
Stateful SD	45 136	55 983	43 214	59 798
Stateful π	30 549	39 067	28 153	33 980
LKH	218 046	269 284	241 362	393 758
Normal SD	222 409	295 778	170 175	305 225
Normal π	153 733	180 157	114 100	180 048

As the diagram and table shows, the stateful version acheives significantly lower bandwidth requirements compared to previous schemes. The worst-case performance (i.e., the maximum number of sets used) is reduced by a factor five and the average case is reduced by a factor four.

5 Security of Stateful Subset Cover

The scheme as presented is not fully collusion-resistant. Users can collaborate by one revoked user who has recently been removed sharing the current state key with a user who is covered, but who does not have the state key. In this section, techniques for mitigating this type of attacks will be discussed. Further, a model for this type of cheating will be given and, the expectancy for how long cheaters gain access will be analyzed using the same data as was used for performance evaluations.

5.1 Security Model

In a commercial setting, the concern is to make it cost-inefficient rather than impossible to illegally decrypt the broadcast. In this model it is assumed, that it is possible to make it expensive to extract state keys, by putting them in a protected area such as a smart card or other tamper resistant hardware. In addition, by using periodic hard updates, cheaters will also periodically be removed from the system. These two techniques can together be used to mitigate the effect of a collaborative behavior by dishonest users.

The major threat in a commercial setting would be the extraction of legitimate long-term keys, since that would allow for pirate decoders to be manufactured. This can be made hard by placing the long-term keys in protected hardware, and by using *traitor tracing* techniques, should the keys leak. Concerning this threat, the stateful subset cover schemes presented here have the same security properties as regular subset cover schemes, given that the long-term key structure is identical.

Another important threat would be redistribution of the group key by a member, i.e. every time a new group key is distributed, the dishonest (but paying) member sends the new group key to her friends. The group key is identical for all users in the group, so traitor tracing techniques are not applicable.

In the stateful schemes, another option would be for a dishonest user to instead redistribute the state key (along with the group key) to her friends. The advantage to the cheaters would be that this would not have to be done as frequently as the group key is to be distributed.

An important aspect to analyze is the expectancy of the time a user who illegally gets a state key can recover the group key. In our model, every user who is removed from the system is given the next round's state key and group key, so she can recover the group key for at least one more round. Then, as long as she is covered using the state key, she can keep decrypting, but as soon as she is not covered in a round, she will lose her ability to decrypt the broadcast. The larger this expectancy, the more seldom a traitor would need to redistribute the state key to keep enabling her friends to recover the group key.

This model of cheating simulates a user illegally receiving a state key. It is run over the same simulation data as the performance tests to give a real-world like cover. This also means that some users will be added again before they are successfully revoked by the system. These are ignored when calculating the average.

In the simulation, we measure the average number of rounds users who left in round r could watch the show, given that they were given one key. In the simulations, a hard update is always run immediately after the end of the simulation, i.e. it is assumed that after the last round, all current cheaters were revoked.

5.2 Subset Difference

In Figure 5, the average numbers of rounds a revoked user can watch if she receives that key is shown. The cheating model used is described in more detail in the previous section. As can be seen, even with hard updates done every 160:th round, a cheater can still at best expect to see approximately 10 rounds.

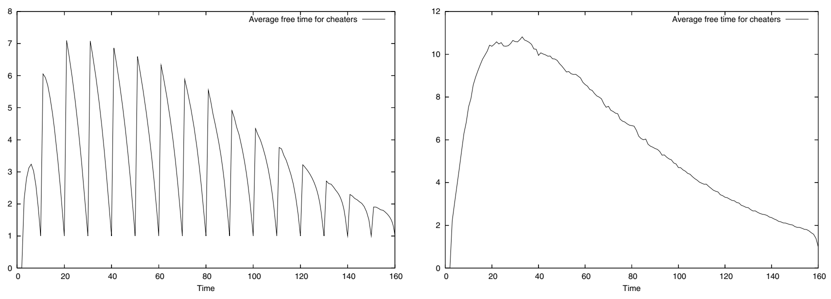


Fig. 5. Average free time for cheaters in stateful SD. 2^{20} users, full-range dataset, basechange 2%, hard updates every 10 (left) and 160 (right) rounds.

While this is worse than the normal schemes, where this number is constantly 1, it is still reasonably small.

At the cost of bandwidth, the frequency with which a traitor must redistribute the state key can be increased by doing hard updates more frequently.

6 Summary

This paper introduces the idea of adding state to a certain class of key revocation schemes, called subset cover schemes. Having state in a key revocation scheme has some drawbacks, like an increased vulnerability to packet loss. These drawbacks are not as bad as they first appear, as we argue for in Section 1.1.

The specific method used in this paper is not collusion-resistant by itself, but may need additional mitigation techniques, such as tamper resistant modules for acceptable security. This non-perfect security is however by practical examples shown to have a limited effect on the overall security from a commercial point of view, where the interest is more directed towards making illegal decryption cost-inefficient rather than impossible.

As a benefit, it is shown that the conversion of a stateless subset cover scheme may lead to greatly reduced bandwidth overhead. This is extremely important in network environments where the available bandwidth is a limited resource, like for example cellular networks. In particular, simulation results show a significant reduction of bandwidth compared to previous schemes.

The transformation presented in this paper is not very complex, with the addition of a global state key, common to all members. Could there be more advanced transformations of subset cover (or other broadcast encryption) schemes which further reduce bandwidth overhead or give better security properties?

References

1. D. Naor, M. Naor, and J. Lotspiech, "Revocation and tracing schemes for stateless receivers," *Lecture Notes in Computer Science*, vol. 2139, pp. 41–62, 2001.
2. S. Berkovits, "How to broadcast a secret," *Lecture Notes in Computer Science*, vol. 547, pp. 535–541, 1991.

3. A. Fiat and M. Naor, "Broadcast encryption," *Lecture Notes in Computer Science*, vol. 773, pp. 480–491, 1994.
4. D. M. Wallner, E. J. Harder, and R. C. Agee, "Key management for multicast: Issues and architectures," Internet Request for Comment RFC 2627, Internet Engineering Task Force, 1999.
5. C. K. Wong, M. Gouda, and S. S. Lam, "Secure group communications using key graphs," in *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 68–79, ACM Press, 1998.
6. D. Halevy and A. Shamir, "The LSD broadcast encryption scheme," *Lecture Notes in Computer Science*, vol. 2442, pp. 47–60, 2002.
7. M. T. Goodrich, J. Z. Sun, and R. Tamassia, "Efficient tree-based revocation in groups of low-state devices," *Lecture Notes in Computer Science*, vol. 3152, pp. 511–527, 2004.
8. T. Asano, "A revocation scheme with minimal storage at receivers," *Lecture Notes in Computer Science*, vol. 2501, pp. 433–450, 2002.
9. N.-S. Jho, J. Y. Hwang, J. H. Cheon, . M.-H. Kim, D. H. Lee, and E. S. Yoo, "One-way chain based broadcast encryption schemes," *Lecture Notes in Computer Science*, vol. 3494, pp. 559–574, 2005.
10. G. Kreitz, "Optimization of broadcast encryption schemes," Master's thesis, KTH – Royal Institute of Technology, 2005.

A Algorithms

CHEAP-FIND-SUBSET(T, v)
Input: T is a stateful SD tree where all nodes up to v have been marked by FIND-COVER()
Output: A subset with top node in v or below, covering all uncovered MC users below v, and as few CC users as possible.

```
(1)   if not $v$.nc and not $v$.cc
(2)       if $v$ = root
(3)           return $S_{1,\Phi}$ //(all users)
(4)       else
(5)           return $S_{\mathrm{par}(v),\mathrm{sib}(v)}$
(6)   if not $\mathrm{l}(v)$.mc
(7)       return CHEAP-FIND-SUBSET($T$, $\mathrm{r}(v)$)
(8)   else if not $\mathrm{r}(v)$.mc
(9)       return CHEAP-FIND-SUBSET($T$, $\mathrm{l}(v)$)
(10)  $excl \leftarrow v$
(11)  while $excl$.mc
(12)      if $\mathrm{l}(excl)$.nc
(13)          $excl \leftarrow \mathrm{l}(excl)$
(14)      else if $\mathrm{r}(excl)$.nc
(15)          $excl \leftarrow \mathrm{r}(excl)$
(16)      else if $\mathrm{l}(excl)$.cc > $\mathrm{r}(excl)$.cc
(17)          $excl \leftarrow \mathrm{l}(excl)$
(18)      else
(19)          $excl \leftarrow \mathrm{r}(excl)$
(20)  return $S_{v,excl}$
```

GENEROUS-FIND-SUBSET(T, v)
Input: T is a stateful SD tree where all nodes up to v have been marked by FIND-COVER()
Output: A subset with top node in v or below, covering all uncovered MC users below v, and as many CC users as possible.

```
(1)   if not $v$.nc
(2)       if $v$ = root
(3)           return $S_{1,\Phi}$ //(all users)
(4)       else
(5)           return $S_{\mathrm{par}(v),\mathrm{sib}(v)}$
(6)   $excl \leftarrow n$
(7)   while not $\mathrm{l}(excl)$.nc or not $\mathrm{r}(excl)$.nc
(8)       if $\mathrm{l}(excl)$.nc
(9)           $excl \leftarrow \mathrm{l}(excl)$
(10)      else
(11)          $excl \leftarrow \mathrm{r}(excl)$
(12)  return $S_{v,excl}$
```

Find-Subset(T, v, *generous*)
Input: T is a stateful SD tree where all nodes up to v have been marked by Find-Cover()
Output: A generous or cheap subset with top node in v or below, covering all uncovered MC users below v.

```
(1)     if $generous$ = true
(2)         return Generous-Find-Subset($T$, $v$)
(3)     else
(4)         return Cheap-Find-Subset($T$, $v$)
```

Find-Cover(T, *generous*)
Input: T is a stateful SD tree where the nodes representing users have been marked. *generous* is a boolean, true for generous cover, false for cheap
Output: A cover $\mathcal{C}$

```
(1)     $\mathcal{C} \leftarrow \emptyset$
(2)     foreach node $v \in T$ in depth-first order
(3)         if l($v$).nc and r($v$).nc and (l($v$).mc or r($v$).mc)
(4)             if r($v$).mc
(5)                 $\mathcal{C} \leftarrow \mathcal{C} \cup$ Find-Subset($T$, r($v$), $generous$)
(6)             if l($v$).mc
(7)                 $\mathcal{C} \leftarrow \mathcal{C} \cup$ Find-Subset($T$, l($v$), $generous$)
(8)             $v$.nc ← true
(9)         else
(10)            $v$.nc ← l($v$).nc | r($v$).nc
(11)            $v$.mc ← l($v$).mc | r($v$).mc
(12)            if $v$.mc
(13)                $v$.cc ← max(l($v$).cc, r($v$).cc)
(14)            else
(15)                $v$.cc ← l($v$).cc + r($v$).cc
(16)    if root.mc
(17)        $\mathcal{C} \leftarrow \mathcal{C} \cup$ Find-Subset($T$, root, $generous$)
(18)    return $\mathcal{C}$
```

TRITA-CSC-A 2011:05
ISSN 1653-5723
ISRN–KTH/CSC/A--11/05-SE
ISBN 978-91-7415-942-4