Enhancing User Privacy in Web Services

by

Jean-Sébastien Légaré

M.Sc. Computer Science, University of British Columbia, 2010

B.Sc. (Hons) Computer Science, McGill University, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES

(Computer Science)

The University of British Columbia
(Vancouver)

November 2017

Abstract

As the wealth of quality application services grows, so too does the volume of data that users disclose and entrust to others. To receive service, users must trust providers with their data. But, without being afforded visibility into the inner workings of the services, users cannot gain strong assurance that their data will be safeguarded.

This trust is precarious. First, services may go out of business or fail, leading to complete data loss, or to the inability to view, manipulate, and share the data in a meaningful way. Second, the confidentiality of user data critically hinges on the service enforcing complex access control rules correctly, and following secure cryptographic practices diligently. Third, even when sensitive data is safeguarded most securely, every application interaction, or communication attempt may still be surveilled and scrutinized, unbeknownst to users.

This thesis presents alternate web application designs that aim to provide users with verifiable guarantees about the longevity of access to data, data confidentiality, and anonymity. We achieve these goals by systematically disaggregating functional components from the service's trusted computing base, and exposing them in a way that allows external verification. Our is aim is to protect personal data, but to do so while maintaining the many advantages that traditional web applications can offer. We demonstrate the effectiveness of our approaches with proof-of-concept applications that are practical and allow immediate deployment in the current web.

Lay Summary

Web users entrust increasingly larger quantities of personal, valuable, and private data to the services they use, such as email, messaging, and social networking. The security of this data depends on the services behaving correctly. Unfortunately this is beyond the control of the users, therefore worrisome: Should I trust a web start-up with copies of my medical or banking history? How can I be certain that data I share now will remain confidential indefinitely? Are my anonymous posts really anonymous?

In this thesis, we propose novel ways to build web services that provide users more accountability and control over data. We demonstrate their effectiveness via practical and immediately deployable applications that offer verifiable guarantees that the data will remain available, confidential, or anonymous. Our applications improve privacy, while preserving the many advantages users have come to expect from web applications, namely universal access, ease-of-use, and effortless updates.

Preface

This thesis combines manuscripts from three separate research projects I have conducted during the course of my Ph.D. program. Despite being compiled in this thesis, each manuscript stands on its own as a scientific contribution. Chapter 2 and Chapter 3, present already published work. Chapter 4, on the other hand, is based on a research paper which is in submission at the time of this writing.

I outline here the chronology between the three projects, and detail the extent of my personal contributions within the total collaborative research efforts that they represent. Readers eager to learn about the main topic of the thesis, may proceed directly to Chapter 1.

The manuscripts have been scrupulously reformatted to conform to the exacting formatting guidelines to which doctoral dissertations are subjected. Only in an attempt to improve consistency, clarity, and logical flow of ideas throughout the encompassing dissertation, have certain elements of the original manuscripts been edited.

Chapter 2: Micasa

Micasa has been published at the 4th Annual Symposium on Cloud Computing (SoCC) in 2013 [38]. I was the main investigator for Micasa for all aspects of the research, and presented the work at the conference. The work was performed under the supervision of Andrew Warfield and William Aiello. The first co-author on the paper, Dutch Meyer, helped with the positioning of the idea, and with the paper's composition. The other paper co-authors have provided crucial help with the implementation of applications evaluating the system, and with manuscript revisions.

Chapter 3: Beeswax

Beeswax was published at the Privacy Enhancing Technologies Symposium (PETS) in 2016 [39]. I was the main investigator for Beeswax for all aspects of the research, and presented the work at the conference. The work was performed under the supervision of William Aiello. Co-author Robert J. Sumi contributed to the implementation, specifically the Application Programming Interface (API) which connects with the Twitter UI, and assisted me in evaluating the system. Arthur Loris contributed cryptographic utility routines.

Chapter 4: DistoRt

A previous version of the DistoRt manuscript is currently in submission. I was the main investigator for all aspects of the work, under the supervision of William Aiello. A predecessor system to DistoRt, Twistor, shared some of the same goals as DistoRt, and is published in Marjan Alavi's M.Sc. thesis [2]. DistoRt was a successful attempt at solving most problems identified with its predecessor, introduced a stronger threat model, and implemented several new mechanisms. The DistoRt co-authors Robert Reiss, William Aiello, and I were involved in the Twistor project as well. A few elements from DistoRt's motivation and related work are also present in Twistor.

The DistoRt codebase is a fork from Beeswax with major modifications: an improved key distribution over Twitter and GitHub, as well as a new cryptographic messaging protocol over Twitter. Alex Ristich contributed code for the GitHub integration and helped with testing and evaluation. Robert Reiss helped with the system design, and with configuration of parameters in the evaluation. Jose Carlos Pazos Ortiz helped shape the system's API and assisted in creating the initial versions of the system's streaming interface with Twitter.

Table of Contents

Abstract . . . . . **ii**

Lay Summary . . . . . **iii**

Preface . . . . . **iv**

Table of Contents . . . . . **vi**

List of Tables . . . . . **ix**

List of Figures . . . . . **x**

Acknowledgements . . . . . **xi**

Dedication . . . . . **xiv**

1 Introduction **1**

1.1 Micasa 4

1.2 Beeswax 6

1.3 DistoRt 9

1.4 Scope 10

2 Micasa: Tolerating Business Failures in Hosted Applications **11**

2.1 Motivation and Overview 11

2.1.1 Challenges 13

2.2 Architecture 15

2.2.1 Micasa Applications 18
2.2.2 Clients 18
2.2.3 Data Stores 21
2.3 Maintaining Service Integrity 26
2.3.1 Data Authenticity and Integrity 26
2.3.2 Data Consistency 27
2.3.3 Validation and Sanitization 27
2.3.4 User Store Abuse 28
2.3.5 Missing User Data 28
2.4 Implementation 29
2.4.1 Before and After Unplugging 30
2.5 Evaluation 32
2.5.1 TwoCans: Messaging System 32
2.5.2 HotCRP-P: Permanent HotCRP 34
2.5.3 Client Performance Overheads 35
2.6 Related Work 39
2.7 Discussion and Future Work 40
2.8 Conclusion 42

3 Beeswax: Preventing Data Exfiltration in Private Web Apps 44
3.1 Motivation and Overview 44
3.1.1 A Platform Approach 45
3.1.2 Adversarial Apps and Other Threats 46
3.1.3 Beeswax 48
3.1.4 Overlay Versus Platform 49
3.2 Architecture 50
3.2.1 Beeswax Operation and API 53
3.2.2 Security of Beeswax Key Distribution 58
3.3 Implementation 59
3.3.1 Secure Display of Private Data 59
3.3.2 Integrity of the Page Runtime 62
3.3.3 Privacy Indicator 65
3.3.4 Cryptography and Key Management 66

3.4 Evaluation 68
3.4.1 Functionality and Experience 69
3.4.2 Performance 71
3.5 Limitations 73
3.6 Related Work 74
3.7 Discussion and Future Work 77
3.8 Conclusion 77

4 DistoRt: Anonymous Communications on Existing Infrastructure . 79
4.1 Motivation and Overview 79
4.2 Components, Assumptions, Goals 84
4.2.1 Components 84
4.2.2 Security and Design Goals 86
4.2.3 Threat Model and Assumptions 87
4.3 Architecture and Implementation 89
4.3.1 Key Distribution 89
4.3.2 Anonymity Groups 94
4.3.3 Sending Twists 96
4.3.4 Encrypting Twists 98
4.3.5 Cover Traffic 102
4.3.6 Receiving Twists / Streaming 103
4.4 Evaluation 103
4.4.1 Maximum Rate of Incoming Twists 104
4.4.2 Maximum Client Bandwidth 106
4.4.3 Twitter API Limits 107
4.5 Related Work 108
4.6 Discussion and Future Work 114
4.7 Conclusion 119

5 Conclusion and Future Work 120
5.1 Future Work 121

Bibliography 123

List of Tables

Table 2.1 Common application features, their categorization, and likely replacements in an unplugged application. 16
Table 2.2 Methods in Capability Storage Interface (CAPSI). 20
Table 2.3 Anatomy of a capability. 23
Table 2.4 Micasa Web Applications and their size. 32

Table 3.1 A subset of the Beeswax Page Runtime API. 53
Table 3.2 Alice contacts Bob for the first time and negotiates a shared secret with him using our key agreement protocol. 78

Table 4.1 Information stored in self-signed certificates. 92
Table 4.2 Processing rates of DistoRt messages. 105
Table 4.3 Network usage of one client as the size of the group and the subgroup (listening position) varies. 107

List of Figures

Figure 2.1 TwoCans, a typical Micasa application. 16
Figure 2.2 Alice shares data with Bob. In this example, Alice's application is interacting with both Alice's and Bob's data stores. . . 24
Figure 2.3 Cumulative distribution function (CDF) of the percentage load time overhead versus the static baseline, compared on a page-by-page basis. 37
Figure 2.4 `getblob` overhead. 38

Figure 3.1 Division of the Beeswax platform into components. 50
Figure 3.2 The Privacy Indicator. 66
Figure 3.3 Median time spent in each component when encrypting DOM elements, by plaintext size. 72

Figure 4.1 Organization of anonymity group `#T` into subgroups (4 levels), and participation of two members of the group. 96
Figure 4.2 Screenshot of a twist on Twitter. 97
Figure 4.3 Message structure of a twist, encoded in the payload of a tweet. 99

Acknowledgements

"Last thing I remember, I was running for the door.
I had to find the passage back, to the place I was before.
'Relax' said the night man, 'We are programmed to receive.
You can check out any time you like, but you can never leave!"'
— The Eagles, *Hotel California.*

My doctoral journey has been a long and winding one, but one filled with excitement. I am now very happy to have made it through. The Ph.D. process is sometimes compared to climbing a mountain, or a gigantic slippery slope. While the analogy is apt – research requires dedication, effort, and perseverance – it abstracts away important details, i.e.,, that as a researcher you are often able to design your own rock face, and that all the routes you choose must be novel. A closer metaphor, in my opinion, is the one of riding a roller coaster in which you have to lay the tracks ahead as you go. There will be suspense, exciting twists and turns, velocity, and a few necessary drops. It is a lot of work, and in the end, it only gets you as far as you choose to go, ideally with more potential than you started with[1].

I can't say I knew that I'd end up working on web services and privacy when I started the program. As I look back at my work, I am satisfied and content to have developed solutions to important research problems in that area. I feel privileged to have had the chance, i.e., the environment, the time, the skills, and the support, to chip away at problems that were personally affecting me, while at the same time getting recognition from my peers.

[1]The pedantic reader will point out that a roller coaster normally ends with no change in potential energy. Here, I am referring to a different kind of potential.

It is fair to say that one can't pull through Ph.D. alone. I want to thank my supervisors, Bill and Andy, for imbuing in me the technical knowledge and critical thinking necessary for academic publication. Thank you for the countless hours of board-drawing, idea-sketching, idea-writing, and perhaps most importantly, idea-shredding (given how easily my interests swivel around my curiosity). Thank you also for entertaining the same morale-boosting pep talk over and over. My intuition is that I could not have succeeded without your help, but I am grateful I did not have to run an experiment to gather evidence of that.

I want to thank also the other members of my examination committee, Ivan, Norm, Sathish, and Roxana, for their precious suggestions to improve my dissertation.

My friends, *mes amis*, it is finally over. Thank you for your unconditional support for the several years during which I was "months away" from finishing. Thank you for creating precious moments away from computers to help recharge my battery, and for taking me on opportunistic adventures around the globe. Also, Patrick, Mihir, *je vous dois une fière chandelle*, for your uplifting, weapon-grade sarcasm and cynicism, lighting up the darker years of my X410D experiment.

Collaborators

Publications, especially in systems, are the culmination of an evolution of ideas and techniques over the course of several conference or journal submission attempts. From one submission to the next, ideas are formed, dropped, reused, refitted, or even kept identical, but motivated differently. A publication's author list is a very limiting form of accreditation for work done. Distilling years of team effort to a simple ordered list of names represents a drastic reduction in semantics. The list of names also do not provide a real opportunity to highlight efforts of unpublished work that happened before a successful publication.

I seize the opportunity here to highlight the efforts of NSS lab colleagues which have influenced the work presented in the thesis, or have contributed to previous projects that have constituted some of the foundation for my work. I couldn't have done it alone.

Thanks Mark, Kalan, alex t., and Long for helping out with Pando (my earliest privacy-related project). Despite not having been successfully published (yet),

conducting this project has allowed me to gain solid experience with browser technologies, experience that was instrumental in starting and completing my following research projects.

For my first successfully published paper, Micasa, I want to thank Dutch, for helping with the art of composition. Framing ideas and system objectives successfully past a program committee is overwhelming at first, so it's great to trade recipes with a seasoned student. Thanks to Brendan and Jake for their contributions on earlier versions of the codebase, and to Mark, alex t., Sara, Kalan, Robert, Quinlan and Dennis for their crucial help with implementation of the so many applications and tools we ended up building in the evaluation.

For Beeswax, thanks to Robert for precious help driving the evaluation, and to Arthur for starting a set of cryptographic utility routines that I later expanded for Beeswax and DistoRt. I take my hat off to Bill, for helping me pull through that one on many submission deadlines.

For DistoRt, thanks to Marjan and Robert for help in taking a first cut at the anonymity problem in Twistor, to Jose and to Alex R. for the camaraderie and helping me ensure everything operated smoothly in the field. Bill, again, a final thank you for showing me how to lift cryptographic weights. That skill turned out to be invaluable in laying down my Ph.D. tracks.

Lastly, I thank you, the reader, for taking the time to read my dissertation. Hopefully, you will enjoy it.

À ma famille, pour votre soutien inconditionnel,
votre confiance en moi et vos encouragements.
Je suis heureux d'être parmi vous.

Chapter 1

Introduction

The transition from software installed locally to Internet services has several appealing benefits. To name a few, these services provide the convenience of rolling automatic software updates, the so-called "frictionless" sharing between users, and a 24/7 quasi-universal instant accessibility. It is no surprise that we see ourselves investing increasing amounts of our lives online. Canadians, for instance, are amongst the most engaged users in the the world in this regard, spending on average 36.7 hours online per month [85]. With the ubiquitous availability of mobile broadband, users can be online on-demand, at all times.

The web model allows service providers to abstract away all of their business logic and hardware behind a simple web API. For users, this simplicity provides convenience. It means, namely, that little effort is required from users to try out a new application. The wall of separation between users and the servers can also be beneficial for the provider. It procures a facade, a technical advantage to protect proprietary technology, and provide transparent scalability.

But this separation is also cause for serious concern: users have little to no visibility into what goes on beyond their computer. Consequently, and with reason, users worry: Will my online documents still be online in ten years? Should I give my credit card company access to my email account if it means lower annual fees? Should I trust a new web start-up with copies of my medical records or banking history? How can I be certain that the data I share now will remain confidential indefinitely? Will my communications and thoughts I write down be recorded? Is

my anonymous forum post really anonymous?

The modern Internet, unlike any information system before it, insidiously demands sacrifice from users. Online services may be labelled as "free-to-use", but there still is a cost to using the services – in the form of valuable personal information. Uploading data to a service generally implies, unfortunately, losing control over it. While we, as individuals, reap the benefits of the growing opportunities for communication and commerce offered, we are forced to *trust* the safety and stability of hundreds of individual online services to protect our privacy. Not only do users have to trust how services behave now, but also how they will protect their content in the indefinite future.

Web service providers, for reasons that are as varied as the services themselves, will take charge of every aspect of their users data. Namely, they are responsible to host it, to share it, to protect it, and to present it. This is convenient for users, but creates a severe privacy issue. The heart of the problem we are attacking is the resulting lack of assurance for users: without being able to verify the veracity of privacy claims or the security of the hardware and software systems involved, users are left to wager on whether their trust in a service is well-placed.

Our hypothesis is that we can build practical Internet web services that not only safeguard users' private data, but that do so in a manner that is verifiable externally, more specifically, whether user data can be accessed in the long term (accessibility), whether access control is enforced properly when sharing (confidentiality), and whether metadata is recorded (anonymity). By practical, we mean that it should preserve the benefits of a hosted web application, and that it should be easy to deploy in today's web ecosystem.

Our overall approach is to expose aspects of the service responsible for protecting user data, commonly enforced on servers, to the clients, and allowing the latter to verify the exposed functionality. As a result, the users do not need to rely as much on the service for correct functionality as in current web services. For instance, to expose access control policies, which are commonly enforced with access control lists in a server database, our approach is to enforce rules using client-side cryptography, and allow users to verify the policies with client-side tools we provide.

We show on three counts of following this approach, that exposing functional-

ity permits providers to make explicit privacy guarantees about their service, and simultaneously allow clients to verify them. This allows users to make better informed decisions about entrusting their private data to a service. But we also show that the disaggregation and exposure of functionality can be achieved in a way that preserves many desirable properties of current web services, such as rolling updates, easy sharing, transparent server scalability, and easy browser accessibility.

In each of the following research projects, moving functionality to client control results in an improvement to a different aspect of user privacy. Each work suggests a novel point in the design space of web applications, and materializes the design point through implementation. We assess the practicality of each approach empirically whenever possible, and the security, qualitatively.

Modern web services mediate every access to a user's own private data. Our first concern is about users losing access to their own data. Unfortunately, Web applications offer limited guarantees that data will remain available and accessible into the future. Catastrophic events such as hardware failure or corporate bankruptcy may result in services failing forever. Similarly, API changes in services may obscure or eliminate access to data that was previously accessible. This problem is about not only a possible loss in durability of the data (e.g., data being deleted accidentally too soon), but also about the loss of ownership and control over the data – uploading data unfortunately currently equates with transferring control. In our first research project, Micasa, we present the loss of access to user data as a recoverable failure mode, and we develop new web applications that are tolerant of such failures.

Each web service defines different complex rules for privacy and confidentiality. Our second concern is about the difficulty of enforcing security guarantees across a collection of web services, e.g., confidentiality, authentication, authorization. For instance, each application makes different claims about the accessibility of user data (intended for a public audience, for friends only, for individuals, etc.). Unfortunately for users, data confidentiality critically depends on the provider's implementation. Users could conceivably verify that the privacy claims of one cooperating application (e.g., by auditing source code, or by relying on the verdict of a group of experts) are respected, albeit laboriously. However verifying the security claims of the sum of all applications a user may be interested in is prohibitive.

In our second project, Beeswax, we build a platform that can reduce the burden of verifying the security claims of a whole ecosystem of private web services to that of verifying the security properties of the platform alone.

Web services commonly provide messaging, but the semantics of the message delivery are opaque to users. Our third concern is about anonymity. The scale of modern application services has now reached billions of users [93]. As more people must rely on online identities to address each other and communicate, and as worries of mass-surveillance rise, so does the need for anonymous communication. How can a user be certain that a message will be sent or has been received anonymously? To ensure a certain level of anonymity has been attained, a user must be able to control, or at least reason about the network on which its messages transit. In our third project, DistoRt, we observe that in many anonymizing application networks, such as Tor [23], the anonymity critically depends on the security of their network routes. We also note the lack of a existing web service providing a complete solution for anonymous communications. In DistoRt, we build a system for anonymous communication over existing scaled services (rather than over IP), in a way that allows the network to misbehave, without affecting application security. We also reuse existing web services to provide a convenient user authentication and key distribution mechanisms.

Before we delve into full-length descriptions of the three systems Micasa, Beeswax, and DistoRt (Chapter 2, Chapter 3, and Chapter 4, resp.), we highlight the contribution of each one in the context of our thesis.

1.1 Micasa

Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC) 2013
Jean-Sebastien Legare, Dutch T. Meyer, Mark Spear, Alexandru Totolici, Sara Bainbridge, Kalan MacRow, Robert Sumi, Quinlan Jung, Dennis Tjandra, David Williams-King, William Aiello and Andrew Warfield.

Users of hosted web-based applications implicitly trust that those applications, and the data that is within them, will remain active and available indefinitely into the future. When a service is terminated, for reasons such as the insolvency of the

business that is providing it, users risk the immediate loss of software functionality and may face the permanent loss of their own data.

Having witnessed several such events before starting the project, we searched for a way to have functionality of an application outlive its application provider.

A common practice of providers is to offer users a "check-out" of all the data contained on the application account: before an application provider closes its doors, users are generally given the "chance" to export all (or most of) their data to an archive. If in luck, then the archive is in a format that is supported by another application, and can be subsequently re-imported. In this case, the loss simply amounts to the time investment in familiarizing oneself with the original application. Odds are, however, that the data is application-specific and cannot be re-imported elsewhere. Software functionality is much needed to search, analyze, present and share the data in a meaningful way. Without it, the exported data is essentially useless.

We have developed a web platform, Micasa, which allows developers to build applications in which private data (posts, images, messages, etc.) remain accessible after provider end-of-life (EOL). The platform offers ways to continue to deliver *data*, and preserve *functionality* to access this data, *after* the provider stops supporting the application. The Micasa platform distributes authorization decisions between user-chosen third party storage providers, and caches application logic on the client side. Together, distributing user data and storing logic allow the possibility to enjoy application functionality with user data in perpetuity.

Micasa applications are partitioned by developers into server- and client-side components. Client-side application logic, written in JavaScript and HTML5, is stored alongside user data in a third-party storage service, chosen by the user. Users being able to choose a location to store their data is one of the ways in which they can conserve control over access to the data. The third-party storage service runs a specific capability-based protocol, which can enforce authorization in the absence of the provider. This allows social features (e.g., sharing or collaboration) to persist after provider end-of-life (EOL). The goal is to make the application software provider critical in the least number of ways possible.

Under normal operation, the provider is responsible for maintaining central information (e.g., the list of registered users), private user data and computation-

ally demanding functionality. However, in the event that the provider is no longer available, the application is capable of continuing to offer a subset of functionality, using only the client-side code and associated storage services. To ensure that the exposure to lost functionality remains explicit, Micasa includes a browser-side monitor that audits RPC interactions with hosted components of the application, and also allows users to "unplug" applications to simulate provider EOL. The platform thus allows users to quickly and reliably verify the claims of the application provider.

Micasa ships as a two-tier application library consisting of a JavaScript runtime to provide support for the data-access protocol, and a Chrome extension to monitor and intercept the application's network requests. Requests are grouped in different categories based on their URLs, and to each category a policy decision to either block or allow the connection. The policy decision is based simply on whether the application is currently "plugged" or "unplugged". For instance, when unplugged, requests to the provider are disallowed, but third-party requests are still allowed.

By using Micasa, an application service provider may offer additional guarantees to its users vis-a-vis the capability of certain application features to operate independently of the provider's environment. Users no longer have to trust application service providers to provide those features. We believe offering the guarantee to be able to access your data in perpetuity reduces risk for the user and could, for instance, attract new users to up-and-coming applications. The platform monitor allows users to verify claims that could not be verified before: the monitor allows users, at any time, to simulate the absence of the provider and verify that the functionality is maintained. Furthermore, applications written with the platform can still enjoy the benefits of traditional web applications, such as ease of adoption and rolling updates.

1.2 Beeswax

Proceedings on Privacy Enhancing Technologies (PoPETs) 2016
Jean-Sebastien Legare, Robert Sumi and William Aiello.

Users of hosted web-based applications also have to implicitly trust the provider for safe-keeping the confidentiality of their private data. To incent use of their

services, especially for high confidential data, application providers will often advertise their privacy-friendly policies. For example, ChatStep [84], niltalk [107], Google Hangout's "off-the-record" conversations [103], and Slack [122] claim that the data that users pass through their service will not be stored or will be stored but not monetized.

Since such claims are enforced on the server side, the only assurance that users have is the knowledge that providers have a strong financial incentive to avoid a public breach of these claims. On the surface, web applications like noteshred [108] (encrypted notes and messages), Subrosa [123] (encrypted chat), and Cryptocat [88, 89][1] (encrypted chat) provide more assurance of confidentiality as they use client-side cryptography. However, after having been lured to an application by promises of privacy, how would a user know that the client-side code was not exfiltrating private plaintext or keying material?

In Beeswax, we are concerned with the choice users have to make with regards to the confidentiality of their data: to either not use a application that deems itself private, or to use it, but accept that data may be exfiltrated by this application. Static or fully client-side applications could be verified, albeit with difficulty. Hosted web applications, on the other hand, are practically impossible to verify, given their highly-dynamic nature. Static applications, or applications that employ client-side cryptography can improve security, but it remains that each application has to be trusted independently, e.g., each application might employ cryptography or store keys differently, or carry its own unique set of vulnerabilities. Both the verification process efforts, and the total trusted computing base (TCB) are multiplied by the number of applications a user may wish to adopt.

We have created Beeswax, a platform for building new confidential web applications. The platform achieves a two-pronged goal: 1.) to allow for applications that have rich interactions with service-side functionality, and 2.) to reduce the assurance of a whole ecosystem of private web services to the assurance of only the platform code.

In our proposed approach, the responsibility for the overall functionality of

[1] Cryptocat was initially written as a web application, rewritten as a browser extension around 2012, and then rewritten for desktop in 2016. Forks of earlier versions are available online, one of them being Cryptodog.

a service is split between our client-side platform, Beeswax, and the code of the service provider (both client- and server-side). We reduce the total TCB of applications by enforcing access control using cryptographic keys that we store in the Beeswax platform. The application communicates access control policies to the platform via a narrow API, but the cryptographic operations in the platform ultimately enforce them. Users do not have to use these APIs, only the UI provided by the application's client-side code. For instance, a user might click on a friend's name to invite them to a calendar event, which would cause the application to use the APIs accordingly. The platform is responsible for managing keying material, including their distribution, and isolating protected content from the application. The application is responsible for the rest of the functionality, including its look and feel, user-controls for sharing, and distribution of (encrypted) content.

The platform provides its API to every web page the user visits. One particular aspect of the implementation is that it allows users to see content in plain view, while the application only has access to ciphertext. A large portion of the platform API concerns sharing, i.e., the definition of access control lists on data items. Via this sharing API, an application can for instance declare a user message intended for a user Bob. From that point on, the platform enforces the confidentiality of the message by hiding cryptographic material and plaintext from the application code. The platform also provides abstractions and protocols to exchange keys between users transparently, i.e., requiring minimal effort from the application provider.

Beeswax denies the application code direct access to keys and plaintext, but still allows the application flexibility to control both sharing policies and the UI (presentation of confidential data), which allows for traditional development practices. Even with this flexibility, the correct behavior of the application is not critical to the security of the entire system: a malicious application written with Beeswax *could* misbehave, and it would still not compromise the security of the platform or user data. Similarly to Micasa, the platform provides a monitor which allows users to verify the Access Control List (ACL) rules chosen by the application. This monitor forces the application to be upfront and honest about its claims of confidentiality.

1.3 DistoRt

Jean-Sebastien Legare, Alex Ristich, Robert Reiss, Jose Carlos Pazos Ortiz, and William Aiello. (In Submission)

Online anonymity and confidentiality are highly desirable properties in a communications system, but extremely difficult to attain. We wish to obtain the simplicity and availability of web services into a system that respects user anonymity and data confidentiality. We present our application DistoRt, which allows users to form *ad hoc* anonymity groups overlaid on existing scalable services, namely Twitter and GitHub, using existing account identities.

DistoRt mitigates against two weaknesses observed in existing anonymous communication systems. First, many systems provide no key distribution, and leave users vulnerable to traffic analysis in their *ad hoc* attempts to collect the keys of the parties with whom they wish to communicate. This applies to prior work on mixnets, e.g. Vuvuzela [67] or Stadium [64], and p2p networks like Tor. Identity-Based Encryption (IBE) (e.g., Alpenhorn [37]) is a cryptographic mechanism for alleviating the problem of key distribution, but it forces users to trust a key service with master keys instead. We note also that, with the exception of Tor, existing anonymity-friendly networks have not yet gained wide adoption outside academic circles. In DistoRt, we provide a careful protocol for key-distribution, and one that is usable today.

Second, in systems that rely on multi-hop routing, it is difficult for the typical user to reason about the unlinkability between the sender and receiver purportedly achieved by a random set of hops with varying trust levels. In the case of Tor, for instance, the security of the communications hinges on a sufficient number of routers not being compromised, amongst other things. This is true also of most of the prior work on social network routing overlays.

In DistoRt we take advantage of the scalability of Twitter as a broadcast medium to simulate a "one-hop" anonymous network node to which all users are connected. We use cryptographic primitives to fulfill what would normally constitute routing – whether a user is the intended recipient of a message is determined by the ability to decrypt correctly. By simplifying the structure of the network, we can reduce the number of elements to reason about, and consequently the amount of trust need-

ing to be placed in the network. There are drawbacks in relying on broadcast to disseminate messages, as it incurs additional resource usage. However, our resulting system is immediately deployable and can scale easily to groups with tens of thousands of users on modern consumer-grade hardware.

In DistoRt, Twitter plays the role of both messaging service and key distribution service, placing it in a position to equivocate and man-in-the-middle user communications. To thwart this possible attack, we introduce GitHub as a second key distribution service, which again, allows one more entity to misbehave.

DistoRt allows clients to reason more easily about the anonymity property of their communications in the application. Reusing existing web services for overlays confers several advantages. It provides immediate deployability without requiring new infrastructure, and it conveniently allows users to be authenticated using well-established identities (e.g., Twitter handles).

1.4 Scope

Throughout the thesis, we focus on solutions that are technically feasible, deployable, and enhance the privacy of users. We recognize that a feature that would be beneficial for the privacy users may not necessarily align with the motives of *all* service providers. We understand that, for instance, providers dependent on monetizing user data may find some of the tools presented in this thesis less appealing. We do not claim to fully cover all web service models.

Sound services can be built with the technologies presented in the thesis. Despite not covering all business models, we believe that there are incentives to adopt our systems. Many providers fully recognize the importance of privacy, anonymity, and long-term availability of data, to the point of actively using them as promotional advantages for services [80, 81, 96]. We note that in many cases, privacy and business tradeoffs are possible. In particular, the adoption of Micasa or Beeswax to protect user content does not preclude the possibility of advertising based on URL traversal or page visits. We conceive that providers may wish to make stronger guarantees about their users data, guarantees that can be verified. Our solutions will appeal to those providers. And equally, these providers will appeal to privacy conscious users.

Chapter 2

Micasa: Tolerating Business Failures in Hosted Applications

> "On July 3, 2012, picplz will shut down permanently and all photos and data will be deleted. [...] Thank you for your support of picplz and we apologize for any inconvenience this may cause you."
>
> *—Message received by the users of picplz, an (insufficiently) well-financed photo sharing app, on June 1, 2012 [112].*

2.1 Motivation and Overview

Will the cloud-based applications that you use today still exist in ten years? What would you lose if they were to discontinue service tomorrow?

As a growing amount of the software that we use—both as individuals and as organizations—is offered in the form of hosted services, questions like these demand careful consideration. Application hosting is a competitive and operationally expensive market, and provider business models do not always prove to be sustainable. As has already been the case with a number of real systems, the abrupt application end-of-life (EOL) that follows the decision to discontinue a given service

risks the loss of both software functionality and user data [94, 99, 101, 102, 112].

Interestingly, this exposure to risk is not a necessary property of hosted applications: the consolidation of application logic and the storage of user data within an application provider's servers is simply the way that systems have been built in the past, and is a model that is supported by most popular development frameworks. Moreover, building a large-scale hosted application is a challenging problem unto itself, and providers have understandably chosen to invest efforts in developing and scaling their own applications rather than providing features that anticipate their own demise.

We believe that the risk presented by application EOL is significant. As application markets evolve over the next decade, it seems very likely that additional applications will cease operations, resulting in inconvenience and potentially even considerable expense for users. As a result, users may hesitate to invest time in new applications, and organizational software procurement processes may place priority on established and incumbent applications. The perceived risk of using a new service will further challenge the ability of new entrants to innovate and succeed in the application marketplace.

In this chapter, we describe *Micasa*, a web-based application runtime that treats the sudden and permanent unavailability of an application provider as a recoverable failure mode. Micasa makes the trust that users are placing in an application service provider explicit, by allowing large portions of application data and functionality to operate *independently* of the provider's hosting environment. Our system aims to find a balance that preserves the benefits of today's hosted applications—including the ease of adoption, maintenance, and software upgrading—while allowing providers to clearly demonstrate to users that their data and a relevant subset of application functionality will remain available in perpetuity.

Micasa applications are partitioned by developers into server- and client-side components. Client-side application logic, written in JavaScript and HTML5, is stored alongside user data in a third-party storage service (such as Amazon's S3), chosen by the user. Under normal operation, the provider is responsible for maintaining central, private data and computationally demanding functionality. However, in the event that the provider is no longer available, the application is capable of continuing to offer a subset of functionality, even "social" features requiring

interaction with data owned by other users, using only the client-side code and associated storage services. To ensure that the exposure to lost functionality remains explicit, Micasa includes a browser-side monitor that audits RPC interactions with hosted components of the application, and also allows users to "unplug" applications, simulating provider failure.

While some hosted applications have provided interfaces for users to "take out" [124] their data, the result is generally a large volume of JSON- or XML-encoded data, leaving no mechanism for users to usefully interact with the contents. Further, as data representations and schemas may change over time, writing third-party tools to interact with these backups has proven to be a challenging task [78]. By packaging application logic for data access and presentation alongside user data, Micasa ensures that user data is preserved in a manner that is more likely to be usable immediately upon EOL and that can be preserved, in an archival sense, for long periods into the future.

Micasa takes advantage of rich, browser-side execution environments and user-facing storage services in order to achieve a clearer degree of trust between users and application providers. While it does not protect the entirety of application functionality in the event of EOL, we believe it is a useful first step. In particular, the risk mitigation enabled by Micasa allows upstart providers to make clear claims to potential customers about service longevity even in the face of EOL, which provides a competitive advantage over services that cannot (or choose not to) make similar claims.

2.1.1 Challenges

New applications written with Micasa can provide users with a clear guarantee of both features of an application and the set of their own data that will remain available even after EOL. Our system seeks to preserve the scalability, availability, and performance goals of today's centralized application models, without entrusting a single fallible entity with the hosting of data and application logic. Our approach is to move user data out to external cloud storage services and create an access path to this data for the application. Many characteristics of hosted applications make this decentralization difficult:

Single point of authority and control. Centralized control services, accessible with a well-known identifier (DNS name or URL) act as a rendezvous for client browsers that are unable to communicate directly with one another.[1] This control service updates clients on every visit, enforces authentication, authorization, input validation, and serialization of requests as per the desired application policies.

Proprietary information hiding. Centrally hosted applications provide a convenient location to store data invisible to clients, such as the exhaustive list of registered users, algorithms, and keys.

Scalability. Centrally hosted providers benefit from elastic scalability within a single operational environment. A provider can use cloud computing to grow, shrink, and relocate their compute power to adapt to changing user demand and maintain suitable performance levels.

Global view. Centrally hosted providers benefit from a global view on all data in the system. This is useful for building fast search engines, spam detectors, and enforcing constraints across all data (e.g., uniqueness of user email addresses). Finally, storing all data centrally and controlling access to it allows application developers to decide on storage formats and infrastructure, and evolve them over time.

Micasa applications are distributed by a central hosting provider. When the provider is available, the application benefits from all the advantages listed above. Unlike traditional applications however, Micasa applications can preserve core functionality in the event that the service is discontinued. When this occurs we say that the application has become "unplugged".

Micasa eliminates the need for the application provider to mediate access to user data and protect data integrity. However, unplugged applications are not exact analogs of today's centralized applications—we do not attempt to distribute proprietary information, nor preserve a global view on all data.

Micasa will support certain classes of applications better than others when unplugged. Applications which are heavily based on individual user-data-driven views such as blogs or photo galleries are the easiest to support. With Micasa's

[1] There are upcoming browser peer-to-peer technologies, but they require addressable proxies to establish connections.

data interface, applications can share data objects between users, and support user comments and ratings (TwoCans, in Section 2.2, is an example of this).

On the other hand, Micasa is less suitable for applications that rely heavily on proprietary or global data, e.g., a web search engine, or a matchmaking dating site. There is still value in using Micasa for these applications however, because Micasa allows archiving, indexing, and searching both the content in a user's personal store, as well as content shared by the user's "friends". For instance, a hypothetical Netflix-Micasa application might not offer recommendations when unplugged (because computing those might require ratings of all users), but still allow an individual user to look at (or search through) the list of all the previously viewed items and ratings in their social graph (HotCRP-P, in Section 2.5.2, covers such archiving and search).

Our implementation focuses on web applications, and therefore we limit unplugged operation to computations supported by modern web browsers. Also, we do not offload any of the application logic to storage-side services, aside from access control checks. This implies, for instance, that a webmail service built on Micasa could support, once unplugged, access to inbox contents and sharing of messages via our sharing API, but not reception of email via SMTP from other mail servers.

Applications that serve cached pages for high performance or that offer notifications to their users, such as Twitter or Flickr, can still do so while the provider is present. After EOL, Micasa can offer viable fallback modes of operation. For instance, notifications can be replaced with polling, and caches can be replaced with direct access to user data stores (Section 2.5.3 covers the caching example). Other examples of unplugged functionality are summarized in Table 2.1.

2.2 Architecture

We will explain Micasa's architecture through a motivating application example. Figure 2.1 is a high-level architecture diagram of *TwoCans*, a shared chat and messaging system similar to services such as Google Chat, Google+ Hangouts, or Facebook Chat. Its design is representative of a typical Micasa application. TwoCans is normally available as a browser-based application from a provider at a well-known

		Converted feature	
Feature Name	Class. (§2.1.1)	Unpluggable?	Solution Summary
ACL / Confidentiality	single-point	yes	Client-side crypto. Users store encrypted blobs and metadata. Group keys can be shared with closed caps.
User Registration	prop. info. hiding	lost	No new users can register. Known registered users can be remembered however.
Content Discovery	global view	lost	Requires access to global data. Limited form of discovery possible out-of-band (e.g., URLs in emails).
Notifications	scalability	degraded	Polling for object modification time changes or pub-sub mechanisms implemented with append operations.

Table 2.1: Common application features, their categorization, and likely replacements in an unplugged application.

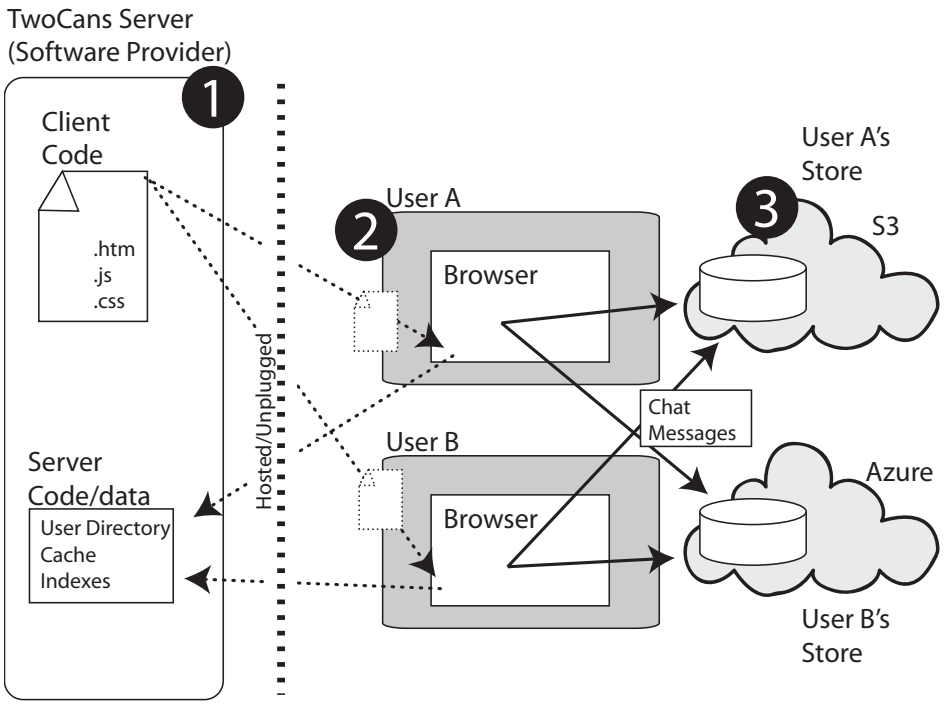

Figure 2.1: TwoCans, a typical Micasa application.

URL. However, if the application is ever discontinued, TwoCans still has access to chat histories, as well as the ability to interact with existing known contacts. A key requirement to enable this post-EOL functionality is that user content not be stored by the central server, or else it could disappear with it.

The software provider distributes the TwoCans source code from their servers, labelled as (1) in Figure 2.1. We discuss how this Micasa application differs from a traditional web application in Section 2.2.1. Label (2) shows this same code running inside the user's browser, where it links with the Micasa library, which we discuss in Section 2.2.2. This library includes a secure monitor, which ensures that the source code abides by the rules for unplugged applications. A code cache is also available locally at the client, for both the application code and the library itself.

In Micasa applications, users provide the application with the routable name of a Micasa-compatible data store of their choice (e.g., during registration), denoted by Label (3) in Figure 2.1 for TwoCans. Users store personal content on their chosen data store and retrieve the content of other users from their respective data stores. That is, users do not interact directly as in peer-to-peer systems nor do they interact using a service provider as a relay. Rather, they interact using the personal stores as intermediaries in what we call *peer-to-store* communication. The programming interface of the personal data store is discussed in Section 2.2.3.

Users could in principle run their own storage server on a home network. However, we assume that most users will use a commercial storage provider for their Micasa applications to take advantage of the durability, availability, and reachability of a commercial provider. A user may, but need not, use the same storage provider for every Micasa application.

By using third-party storage we are not simply exchanging dependence on one service for another. Internet-scale storage is now mature, highly reliable, and revenue-generating. The cloud storage business model depends on protecting the integrity of the data stored, and providing users with the ability to retrieve it. While dominant cloud storage providers have proven thus far to be stable and lasting service offerings, Micasa still allows application code and data to be migrated from one storage provider to another. We assume that if a large storage provider were to go out of business, it would provide its customers with sufficient time to perform

migration. We discuss data store migration in Section 2.2.3.

2.2.1 Micasa Applications

A chief challenge in Micasa is to survive the failures associated with provider end-of-life without sacrificing the many benefits conferred by centralized, cloud-based application architectures. In particular we wish to preserve performance and availability at scale. Rather than attempt to replace existing application models outright with a peer-to-peer architecture [91], our philosophy is to embrace the same core approach that centrally hosted applications use today, but endeavour to remove the availability of application providers themselves as a central point of failure.

The client-side code of Micasa applications makes heavy use of dynamic HTML changes and modern browser features from HTML5, such as CORS [129], to fetch resources from multiple third-party services. Like traditional web applications, the application provider serves the client-side code, and maintains global application data—data that is not owned by any particular user or group of users. Providers may also cache some user data, to accelerate certain operations.

Our model introduces an additional role for a storage service provider, which is to guard access to a user's authoritative copy of their data. A disconnection from the application service provider does not affect this role.

The monitor shipped with the Micasa library allows users to check whether application functionality remains available in the absence of the service provider (discussed in Section 2.2.2). We expect users will find this even more useful than a simple data check-out feature, and not take this as a sign that the service provider expects business failure.

2.2.2 Clients

Clients have access to a local code cache modelled after the HTML5 offline cache, that can be updated by simply visiting the application provider's website. This cache is periodically synchronized to the user's per-application personal data store so that it persists on durable storage.

Micasa applications require our client-side library, libeol, to be installed in the browser. libeol provides a JavaScript API, called Capability Storage Interface

(CAPSI), for interacting with user data stores from client-side code. Our user data store API is described in Section 2.2.3. libeol also monitors network access to both the application provider, `tc.example.org` in the example, and storage providers. The installation of libeol simply consists of registering a new browser extension—only a few clicks are required.

Except for the presence of libeol, TwoCans possesses all of the regular functionality and appearance of a normal web 2.0 application: user actions in the view can issue RPCs to the provider to retrieve additional dynamic content (e.g., search and forms).

This form of deployment follows typical web navigation paradigms and makes trying out new applications very easy. We believe that Micasa applications could also be packaged in forms compatible with current browser "hosted application or local app" concepts such as Chrome Apps, Mozilla Apps, or extensions [97, 98, 136].

Connecting to Data Stores

Micasa's client library exposes CAPSI, which is used to request any data to and from Micasa data stores. Table 2.2 lists its methods, grouped by category. CAPSI allows users to create isolated stores for each application, called namespaces. A user can gain write access to one of the namespaces he or she owns by logging into the data provider from the application associated with that namespace. A session is only valid within a single namespace, for that client, on that application. Once an authenticated session is established with its storage provider, a client can create, manipulate, and share objects from the namespace with other users. Clients can also read or append objects from other storage providers, *unauthenticated*, provided that a valid reference to that object is presented. This way, no extra login procedures are required to access other user stores.

Library Installation and Audit

The client library places a monitor around the application code, which intercepts, classifies, logs, and possibly blocks all external requests issued. To audit an application's dependence on its provider, users can use the monitor to launch Micasa-

Category	Methods
Writing	`putlist`, `putblob`, `append`
Sharing	`mkget`, `mkappend`
Reading	`getlist`, `getblob`, `getnsroot`
Deleting	`delete`, `revoke`
Accounting	`login`, `logout`, `mkns`

Table 2.2: Methods in CAPSI.

enabled applications in "unplugged" mode. When launched this way, the monitor simulates the absence of the provider by artificially breaking all provider-bound connections passing through the application interface, verifying any of the provider's claims about robustness to EOL.

In order for the library to properly monitor the application, its code is loaded in the user's browser first, before the application starts loading. The early injection allows the monitor to gain access to the unmodified JavaScript environment. The library injects JavaScript code in the top of the page to enable CAPSI functionality, as is done in other JavaScript instrumentation scenarios (e.g., MugShot [46]).

When a network request is intercepted, the monitor consults a file supplied by the application developer. The Interface Definition Language (IDL) file simply contains a set of URL regular expressions that classify outgoing requests (more details on the format is available in Section 2.4.1). The monitor finds a matching entry in the file based on the request environment, and a policy decision is applied based on the entry's type and plugged/unplugged status. The monitor records the network interaction (and decision) into an audit log, and can optionally report the contents of the log back to a web server external to the browser.

This mechanism allows motivated parties with sufficient expertise to validate any warranty claims the provider could make about the application, and thus helps prove the provider's good intentions. For instance, if an application offered a private or local search into a user's own data (e.g., Section 2.5.2), an investigative user could trigger search actions in the UI while unplugged and verify in the logs that the application does not contact a central application server to gather search results.

Personal Search

A central provider is very useful for indexing and searching across all of the data associated with an application. However, if all of the indexing and search functionality is located at the provider and the provider fails, the clients are left without an ability to search even their own data. Micasa makes it feasible for clients to keep a rich index. Clients can maintain an index over their own content with references to their own store, as well as an index over all of the data to which they have been given access. An index mapping keys to capabilities can be built incrementally during the process of retrieving content from other users' stores. In addition, from the list of capabilities in the index, a client can periodically recursively crawl the content to which it has access to update the index for mutable objects that may have changed. We demonstrate the personal search capability of Micasa in our conference management system, called HotCRP-P, which is detailed in Section 2.5.2.

2.2.3 Data Stores

In Micasa, browser applications interact with user data stores. Currently, we require that these data stores export a capability interface, CAPSI. For a given application, different users are free to choose different data stores, and a given user is free to choose different data stores for different applications. While users may manage their own CAPSI-store, we envision a Micasa ecosystem that includes a number of commercial cloud storage providers that export an interface with the following:

Provider compatibility. Users can choose their storage service, and can access stores of other users regardless of their chosen provider. The data should be migratable, so that users can change providers.

Sharing and access control. Users can create data objects and share them with other users. References to objects can be used to control access to the information, and encryption can be used to protect the confidentiality of the data.

Revocation. Users can revoke access to a previously shared object. This is the opposite of sharing.

Write access control. To protect the storage footprint of user data stores, we disallow arbitrary writes to data stores of other users. We do, however, provide a

permissions mechanism which emulates append-style semantics for applications. Given the appropriate permissions, one user may append a reference to an object they own into a list of references in another user's data store.

Fine- to coarse-grained sharing. Users can share an individual object, or group multiple objects into collections and then share them all at once.

Unfortunately, the permissions interfaces exposed by existing cloud storage services such as Dropbox, Azure, and S3, are a poor fit for these requirements: they force data to either be open to the public, or to be shared with a named user on the same storage service. Similarly, their access control primitives make it impossible to safely support append-only semantics. Instead we developed and implemented the CAPSI storage interface. This interface is immediately deployable, for example, as an EC2-based service with an S3-based back-end. As storage services have an incentive to attract more customer data, we assume that if Micasa-style applications became popular, commercial storage providers would add CAPSI as a native interface to their service (or alternatively provide access control primitives that allow a compatible client interface to exist).

Capability Servers

With data stores distributed across multiple independent storage providers, as in our model, using authentication and access control lists to implement the sharing of objects between users would require either numerous user registrations (e.g., a user would need an authentication mechanism for each of the storage providers used by his/her friends) or a trusted third party identity provider. Instead, we have chosen capabilities as the basis of our sharing model. This way, access to objects can be communicated over email, or be embedded inside pages, etc.

Capability Representation

We briefly describe the structure of our capabilities, displayed in Table 2.3. We define two types of objects: *blobs* (files) and *lists* (folders). Lists allow capability grouping and nesting. Both types can be shared with other users. Sharing a capability to an object will share both the object and all of its descendants. For applications that require different types of navigation, such as searching by tags, lists

Field	Description
Object Reference	Type and content address of object
Server Reference	Used to resolve server hostname
Issuer	Userid of object owner
Audience	Userid of users who is allowed access
Root Reference	Cap. from which this cap. is derived
Timestamps	Creation, and (optional) expiry times
HMAC	Signature of the cap. with server's key

Table 2.3: Anatomy of a capability.

store application metadata for each entry that can be used to build search indexes.

The objects themselves are either mutable or immutable. Immutable objects are referenced by their content address (content hash). Mutable objects have a unique name on the underlying data store.

Capabilities describe three types of rights over an object: *OWNER*, which is all-rights, *GET*, which is read-only, and *APPEND*, which allows a controlled form of write sharing (only for lists). We found these to be sufficient to build rich applications.

Other fields are used to reduce the scope of capabilities. Limited periods of validity can be set using the expiry time and creation time fields. Also, capabilities contain reserved fields for issuer and audience to determine the owner of an object, and limit access to some users in cases where the capability server can authenticate the requester (see Section 2.3.1).

A server reference field can be resolved to locate the capability server that manages the object. This indirection allows data migration (Section 2.2.3), and transformation of capabilities into URLs. Lastly, every capability generated by the server includes an HMAC over the properties in the capability tuple, computed with its own secret key, to avoid spoofing.

Data Access

Data in Micasa is represented by capability URLs. These define both the Internet location and access permissions of the data, and have a well-defined format that is

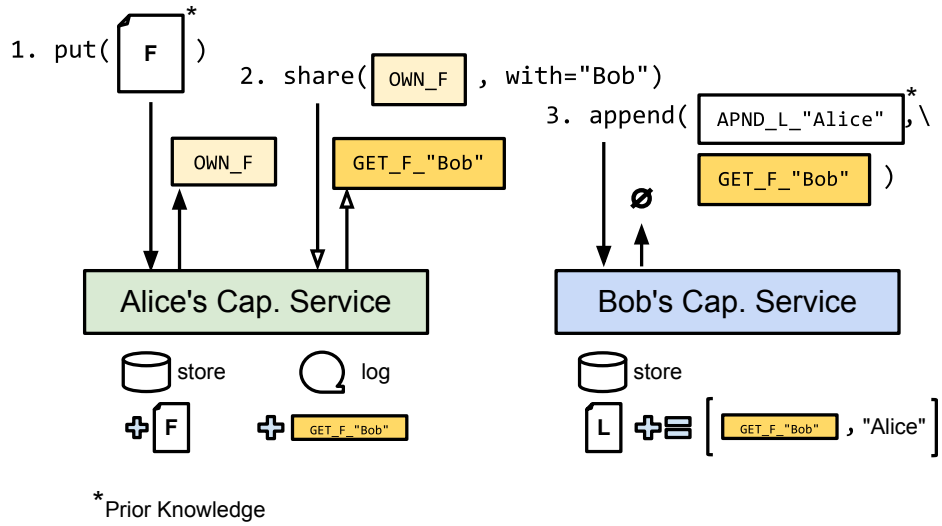

Figure 2.2: Alice shares data with Bob. In this example, Alice's application is interacting with both Alice's and Bob's data stores. Prior to Alice performing a put for `F`, Bob has a list `L` of capabilities in his store and Alice has a capability `APND_L_"Alice"` to append to that list.

understood by all storage providers that support CAPSI.

Figure 2.2 gives a common example usage of our library, in which Alice is sharing some data with Bob. For illustration, we describe this process in the context of a private chat session between Alice and Bob in TwoCans.

We assume Alice gains access to Bob's chat room page by either navigating tc.example.org, or receiving a URL out-of-band. This page contains a capability to the list of messages in the chat as well as an *APPEND* capability to append to the list. To add a message to this chat, Alice needs to first upload a file containing the message to her storage provider. To do so, she types in her message and submits it, which causes the page to upload the file (step 1). After the upload, the response from Alice's storage provider is a capability that indicates that she owns the file. An *OWNER* capability is a proof of ownership of that file and gives Alice read-write-delete permissions over it. The application then converts that *OWNER* capability into a form suitable for sharing (with Bob), a *GET* capability (step 2). The *GET* capability created is logged by her CapServ, so that Alice may revoke it or migrate

it in the future (explained in Section 2.2.3).

Alice transfers this *GET* capability to Bob, by invoking her *APPEND* capability on Bob's server (step 3). A tag parameter (`"Alice"` in Figure 2.2), taken from the append capability, is copied to the new entry in Bob's list, so that he may know which target was used to append. The next time Bob reads the chatroom list, he will detect the new message entry, and can retrieve Alice's file directly from her data store using the capability for that file now sitting in the list.

Data Migration

In Micasa, in order to mitigate service provider lock-in, users store their data with an independent storage provider. We emphasize that by doing so, we are *not* trading one form of lock-in for another. Current storage services have a good track record, and we believe they are less likely to go out of business than many other cloud services. Moreover, the services are effectively built to allow users to simply and easily access and retrieve their data.

In cases where data store migration is desired, Micasa is able to do so while preserving access control and availability. To achieve this, capabilities previously constructed by one server must continue working after a data store has migrated. Because the keys used to sign and verify capabilities are private to each CAPSI storage provider, capabilities issued for objects at the original server cannot be verified at the new server. To migrate to a new server, we first copy all user objects, followed by the capability generation log (one entry for each call to `mkget` or `mkappend` where the capability has not yet expired) to the new server. When the new server receives a request with an unverifiable signature, it can check the log for the existence of a matching legacy capability record, and allow access accordingly.

The server reference field in the capability must always be resolved to the active server that stores the object. Once the data and capability records have been copied, the final step in data migration consists of updating the resolved value of the server reference to point to the new capability service. Our prototype server and client use DNS as a resolution mechanism, but we are investigating other approaches based on email addresses and web discovery protocols.

2.3 Maintaining Service Integrity

For traditional hosted applications, the central servers play a crucial role in maintaining the integrity of the service. For example, they ensure data authenticity, data integrity, and data consistency through validation and sanitizing. Servers also attempt to minimize excessive use of the service (e.g., by bots) or other abusive behaviour. Micasa-style applications do not have the luxury of a central point of enforcement to implement these measures. It is therefore important that the application developer compensate for the absence of a central server by correctly using the tools provided by the library. Below we discuss how this can be achieved.

2.3.1 Data Authenticity and Integrity

Certain situations require proving that objects are authentic to particular users or that access requests have been issued by specific users.

Micasa capabilities are *open* by default, meaning that simply bearing the capability is sufficient to invoke the rights it describes. An optional tag attribute allows differentiating capabilities to the same object, but cannot be relied upon to tell apart two users bearing the same capability.

While authentication could be provided in Micasa through *closed* capabilities, which require the bearer to *prove* his or her identity before a request is executed (e.g., with passwords or email addresses), it is also possible to authenticate requests and data objects via digital signatures, using cryptographic information stored elsewhere in the application. This does require additional key-management complexity, but is less demanding on the capability system, and thus the TwoCans application described in Section 2.5.1 is implemented in this manner.

CAPSI capabilities on their own offer basic support for verifying content integrity. As shown in Section 2.2.3, immutable object capabilities expose the content hash of objects, which can be verified on the client with JavaScript routines or native plugins.[2] Storage providers falsely reporting content can be added to a blacklist and pruned out.

Verifying the integrity of mutable objects is also feasible, but requires informa-

[2] libeol checksums unencoded network payloads (binary) using JavaScript and XHR Level 2 features.

tion external to the capability. In this case, the library provides the tools to digest content and verify signatures, but the application is responsible for providing the expected values.

2.3.2 Data Consistency

To ease application development, CAPSI forces all writes to be isolated and serialized per-object. In our implementation, if the underlying data store can only offer eventual consistency on updates, then updates to an object are logged by the capability service until propagation completes.[3] Updates could also use a version parameter to provide a "conditional put" mechanism.

It is possible to use this consistency model to perform more complex transactional operations, as long as all participating processes cooperate. Unfortunately, this is impossible to enforce if users are not trusted. Our framework is therefore limited to unplugged features that can be supported with single-object atomic operations.

2.3.3 Validation and Sanitization

Applications impose many restrictions on the actions users can take and the data they submit. Coding practices recommend that users' submitted data be validated and sanitized *before* it is persisted. This guarantees that all content on the server satisfies an accepted format.

In unplugged applications however, validating only before submission is insufficient. Because users have full control over their own stores, the data of other users must be verified before it is used. To this end, libeol provides basic common content validators, such as enforcing length bounds on responses and HTML escaping.

The amount of validation needed will depend on the type of application. We have found in practice that non-global uniqueness checks (e.g., a user can only post once on a photo) and chronological checks are simple to implement. Immutability checks (e.g., forbid edits) require chaining digests and are more complicated. Fortunately, expensive validation operations can be short-circuited by memoizing

[3] Propagating updates takes only a few seconds in practice.

content hashes of objects previously visited.

2.3.4 User Store Abuse

Users must pay for bandwidth and space on their storage provider, which represents a new system element vulnerable to abuse. While there are no foolproof solutions, capability servers can mitigate certain forms of abuse.

Rate of access can be controlled by the capability service, through rate-limiting or the insertion of CAPTCHAs. In terms of controlling space usage, users need only worry about other users appending capabilities to their lists; capability strings are relatively small (less than 1KiB), and if this became an issue a maximum length on list objects could be imposed.

It is generally the responsibility of the application provider to rid the application of spammy content and fake accounts. To prevent that sort of abuse, unplugged applications will need to rely on client-side databases and spam engines, or third party spam services. Moderators can still flag inappropriate content, but it is the client-side code that would need to filter it out from view.

2.3.5 Missing User Data

In a centralized hosted service, any revocation or deletion of data by users is mediated by the application. Thus, there will not be any data missing unexpectedly at page build time. In Micasa, however, a user can access and manage data in an application's namespace in his own store via CAPSI out-of-band of the application, such as via a namespace file explorer app. Owners can also revoke access to data that they have previously shared. Capability servers ensure that invoking revoked rights will fail. For the application, this possibility translates into "holes" within pages.

Developers should provide fail-safes for missing content. They should account for the eventuality that content has been revoked, at least at the granularity of the "un-share" operations defined in the application. For instance, if the application allows changing privacy settings on pictures, then access to any picture could be revoked along with all associated information. In this case, the error condition is detected (e.g., 404 Not Found), and the application can replace the image with a

placeholder or another visual indicator.

2.4 Implementation

We have implemented a CAPSI-compliant capability service called CapServ, a client-side library for building Micasa applications called libeol, as well as several applications which will be described later in Section 2.5.

The capability service is composed of less than 3K lines of Python. It is run as a python-WSGI application. It supports three storage backends for data objects: POSIX local file system, Amazon S3, and Microsoft Azure. However, in an ideal deployment scenario, the capability service would be implemented directly by the cloud storage provider.

Our client library, libeol, runs in unmodified browsers. There are two subsystems in the library. First, there is the CAPSI subsystem, which is invoked by the application to access capabilities. This alone is written in approximately 3K lines of Java Google Web Toolkit (GWT) code that compile down to around 120KiB of uncompressed obfuscated JavaScript (33KiB compressed). This subsystem also has bindings for web applications written only in JavaScript (i.e., without GWT). The second subsystem is the monitor, which is divided in two parts. The first part, the in-page monitor, runs in the page's JavaScript environment. The second, the external monitor, runs as a Chrome browser extension.

The in-page monitor is loaded at the very start of each page load. Its responsibility is to bootstrap a communication channel between the page and the external monitor, and provide some hooks that the application can use.

As an extension, the external monitor has the privileges necessary to interpose on and audit all network connections. It captures network events that would be otherwise impractical or expensive to capture from the JavaScript environment, namely network requests triggered by embedding objects in the DOM (e.g., image tags). It also presents a GUI to unplug the application (Section 2.4.1). Cross-browser compatibility is future work, and may benefit from a JavaScript sandbox such as TreeHouse [31].

2.4.1 Before and After Unplugging

In Micasa we leverage HTML5 application cache manifests [132] to identify client-side resources that should be preserved in the absence of the service provider. These application cache manifests can be defined for developers to allow "offline" mode functionality, and can be used as a mechanism to speed up application load times on subsequent visits.

We assume that clients are running our extension when they visit a Micasa-enabled website. Micasa applications are installed to the code cache with the initial visit to a page featuring a special eol marker:

```
<html manifest="man.appcache" eol="true">
```

If the manifest referenced is new to the code cache storage or has changed, then the manifest itself, and all contained resource references are stored. Chrome extensions cannot, as of this writing, programmatically access the browser's application cache contents directly. Our implementation thus introspects the DOM to find the location of the cache manifest and parse its entries. For each entry, a new URL to the corresponding entry in the Micasa cache is created, and a page mapping is updated from original resource URL to cached resource URL. The resources are cached externally to the browser so that they have a longer lifetime than the browser cache.

After unplugging, the local application cache may need to be repopulated from the external code cache. Our implementation redirects web request URLs according to the previously constructed mapping. This is currently achieved with Chrome extensions' `webRequest` module[4]. Programmatic access to the application cache would also be preferable for this task, because redirects performed with this module are unfortunately not origin-preserving for top-level documents.

Manifest Specification

The HTML5 cache-manifests are defined per-page, not per-domain. However, we assume in our implementation that there will be a single Micasa application per domain name, and that the application will be a "single-page" application. This simplification allows the extension to associate one-to-one applications and domain

[4]Proxies are an alternate solution, but complicate deployment.

names, and allows users to unplug applications on demand more easily. We reuse the HTML5 cache manifest syntax and semantics for Micasa applications, except that in order for CAPSI requests to succeed, a wildcard entry must be added to the NETWORK section of the manifest. This entry informs the browser that requests outside the static set of cached resources should be allowed if there is network connectivity.

In addition to the set of static UI resources, the manifest also lists a server IDL file, in JSON format. The purpose of this file is to categorize the requests seen by the application monitor. Entries in this file declare a method name (key), a human-readable description of the operation (an intention), a type label (provider, third party, or CAPSI), and a list of expressions used to match the URLs belonging to the entry. The file is evaluated at load time by the in-page monitor, and its information is communicated to the external monitor.

The external monitor matches outgoing network requests to entries in the IDL file, and will block or allow the connection, depending on the type of the entry and plugged/unplugged status. The default policy when unplugged is to block those requests with type "provider", and those that match no entries (fall through).

The provider may attempt to make bogus claims that certain features are unpluggable, and make them appear to be so via mislabelling. However the monitor provides an audit trail by logging all outgoing requests. A deception exposed in an audit by any customer risks alienating all customers.

The Unplugged Event

UI controls in our monitor extension toolbar allow the user to disconnect from a service provider on demand. This unplugs the application, switches filtering rules in the monitor, and notifies the running application of the state change. Users could use this control to simulate a disruption to provider services, and test that the application features continue to work as advertised.

After failing a server request due to an unexpected network or server error, applications must determine whether the error is transient, or permanent (unplugged). The application code can trigger the application to unplug on its own, for instance if repeated attempts to reach a server all fail. However, to help the application de-

App. Name	SLOC	Description
TwoCans	1500	IM System (§2.5.1).
HotCRP-P	10K	Permanent HotCRP (§2.5.2).
Lenscapes	2200	Photo album sharing.
Data Viewer	650	Namespace file explorer.

Table 2.4: Micasa Web Applications and their size.

cide faster, Micasa defines new runtime page events, `unplugged` and `plugged`, that fire according to the current state of the monitor.

Applications can listen for these events to dynamically change their behaviour. For instance, applications could determine if certain features should be presented to the user, or to pick alternate implementations of a particular feature.

HTML5 offline mode [132] defines similar events for cases where the browser is experiencing a network outage. Our unplugged scenario is similar in that the application service appears unavailable, but different because unplugged clients can still rely on the network to perform CAPSI calls, or access third-party services.

2.5 Evaluation

We wish to create applications that can tolerate permanent disconnection from their provider, but we also want these applications to have good performance, be functional, feature-rich, and practical to build. Those are difficult criteria to meet when considering that the convenience of a central server can be lost. We evaluated the practicality of our prototype by building many different applications, listed in Table 2.4 with their size in lines of client-side code. Of the list, two will be explained in further detail in the following sections. The last section of the evaluation benchmarks Micasa-model applications on a Flickr-based data set.

2.5.1 TwoCans: Messaging System

Our first application showcases the core functionality provided by our prototype library. We use libeol to implement a multi-reader multi-writer system, a pattern common to multiple online social web applications. This pattern often takes the

form of comment lists, page votes, a "friend wall", etc. Multiple instances of this pattern can be duplicated and composed to create pages of arbitrary complexity (e.g., a comment list on a photo, itself inside a shared photo album).

We place this pattern at the core of TwoCans, a multi-party Internet messaging system, written from scratch. It is a conceptually simple application that allows multiple users to exchange text messages in chat rooms. It differs from typical chat programs in that writers retain ownership of the messages they send out. The messages are exchanged between peers using the stores as intermediaries in a peer-to-store fashion. The implementation consists of about 1500 lines of client-side code (excluding libeol), and a small central server implementation of under 300 lines of Python.

To initiate a conversation, users create a list to host the messages. To add a message in the conversation, users first upload message text to their store, then append the resulting capability to the conversation list. The owner of the conversation list can act as a moderator, and revoke access to one of the users. By design, if the owner of a conversation deletes the conversation, the conversation is lost (unless a copy is taken). Also, users can revoke access to their own messages. The TwoCans central server provides discovery of other users and public chat rooms via search. When unplugged, these features go away, but users retain the ability to communicate in chat rooms in which they are already members. Furthermore, inviting users in a chat room can always be done by sharing an invite URL out-of-band.

TwoCans protects against message spoofing via message RSA signatures, generated with utilities in libeol. An author obtains a signature with a libeol function call over the message plaintext, the current timestamp and the unique chat room ID. Signatures are appended along with message plaintext. Other chat members verify them using the author's public key, which is retrieved from the central application server (but could be retrieved from an external service), and cached at the client.

Overall, the TwoCans service consists in a directory of registered users and their public keys, as well as a search index on the subjects of active conversations. Scalability of the service relies mostly on the cloud storage providers of the users.

2.5.2 HotCRP-P: Permanent HotCRP

In this section, we demonstrate that Micasa can support the needs of modern applications, with reasonable developer efforts, by refitting the conference management software HotCRP to work in a Micasa environment. HotCRP-P, our modified application, includes all of the original application functionality, plus the *permanent* ability to search through all reviews and papers the user has ever had access to, regardless of the conference server's availability. The difficulty in this case lies in teaching HotCRP that it can be unplugged, change certain UI flows accordingly, and ensure users can search at all times.

HotCRP consists of about 20K lines of PHP code that mixes data and logic in the page that it renders (like many PHP applications). To change HotCRP into a cacheable web application, we went through the tedious process of extracting UI logic from the server-side HTML generation templates for most of the conference core functionality. The resulting UI resources comprise around 10K lines of JavaScript, 700 lines of HTML, and 200 lines of CSS. Porting an application that from the start embraces Web2.0 further (e.g., no top level page changes, AJAX-heavy) would be easier.

We then modified the login flow in the client to allow connecting the user to a Micasa store, before logging in to the HotCRP server. We modified the paper submission, abstract modification, and review submission flows on the client to work with capabilities (i.e., upload object to store, obtain capability, and submit capability to server). We made corresponding changes on the server so that it stored both the capabilities and a cached version of the full object. Loading these pages as a viewer works in the reverse way: capabilities are fetched from the server via a REST-ful interface (using key parameters such as the paper's id) and objects are retrieved from Micasa stores. If the papers are unavailable, the server's cached copy can be used instead.

We implement the permanent search feature by building a client-side index of page contents, keyed with the paper id of the page being viewed. The index itself is periodically saved to the user's store. Rather than writing our own indexer, we modified Apache Lucene (version 3.3.0) to work in a Java applet. The patch for this is around 1400 lines of Java code. The client stores the capabilities, so that

page content can be reconstructed.

This permanent-access feature is only enabled when the application is unplugged. In that mode, authentication is no longer possible. HotCRP-P thus replaces the login flow with a search query flow. After a successful search, corresponding pages can be displayed, and cached capabilities retrieved.

2.5.3 Client Performance Overheads

We measure the performance impact of migrating from a centralized service to one running in a Micasa environment, in a benchmark modelled after Flickr picture pages. The benchmark consists in rendering picture pages, along with respective user comments, and images embedded in these comments.

We take measurements on multiple picture pages of varying complexity. Pages are representative of a random sample of Flickr picture pages, and were sampled from the "fresh" (recently-uploaded) and "7-day popular" Flickr feeds. The number of comments, the size of the comment text, the images inlined with a comment, the image data, and number of comment authors match the online version of the page in the feeds, and vary across all of the sampled pages. We maintain Flickr's limit of 20 comments shown per page. However, it is not uncommon for comments to embed a variable number of additional images. We run the benchmark on three different versions of the pages, and perform point to point performance comparisons.

The first version is our baseline, a static version of the page with comment text and image tags inlined. That is, all the URLs to images and the comment data are known once the index file is loaded. It simulates an application server that can generate page content instantly (i.e., no page-generation overhead). The server also serves data for all images, scripts, and CSS referenced in the page.

The second implementation uses libeol. In this version, comment and image data are retrieved across CAPSI stores. The base content of the page (HTML, JavaScript, CSS, icons, logos etc.) is devoid of user data. Client-side logic fetches and displays all of the dynamic content in the page from a single top-level capability list for that page. We assign each user involved in the page, i.e., the picture owner and all comment authors, to one store sampled from a population of 10

distinct CapServ stores, according to a Zipf popularity ranking of $1/r$. Users are also assigned a key pair for signing comments. The public key is stored in a key repository on the application server. Individual comments and embedded images are stored as immutable blobs. References to the comments and images are stored inside a list for each comment, and references to these comments are appended to a single comment list owned by the owner of the page. Comments are digitally signed with RSA by including content hashes for comment text, embedded image references, and other fields from the capabilities.

The reference to the appendable comment list, as well as a reference to the main picture is stored inside the top level list. Overall, the client recurses 3 lists of capabilities to obtain references to all objects to be displayed in the page. To verify signatures on comments written by authors unknown to the client, missing public keys are fetched from the key repository, and cached to avoid future lookups.

The third implementation is similar to the second, but benefits from application server caching. Page loading is sped up by retrieving a cache file from a central application service. This simulates an application that caches capabilities added to the page by users (i.e., whenever a new comment is appended). The cache file contains a "flattened" view of the capability structure of the page. It contains capabilities to the user data (comment blobs and embedded image blobs), but not the data itself. In this version, signature verifications on author comments are skipped because the cache file is assumed to be authoritative. However, object digests must still be computed to match expected values in the cache file.

We run all of the CapServ hosts in 32bit Ubuntu 10.04 micro-instance VMs on Amazon EC2 (west coast), and configure them to store objects on their local block storage. Our client runs Google Chrome Stable 25, on a Ubuntu Desktop 10.04 64bit, with a Core i5 750 CPU (4 cores, no hyper-threading), connected to the university's public network.

RESULTS. The bandwidth overhead for any page and version depend upon the number of comments and data objects. The amount of application data loaded is the same across the three versions (images and comments are the same size), so any additional data transferred comes from capability lists, capability strings referring to data objects, and user keys. The two versions using Micasa have the advantage

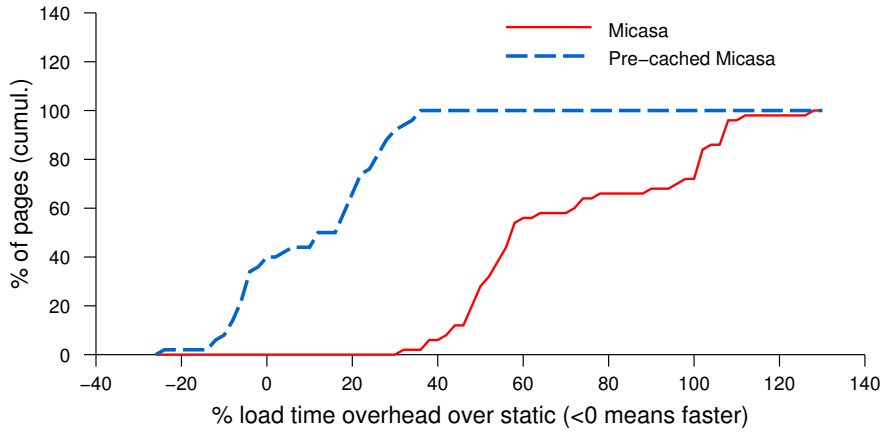

Figure 2.3: CDF of the percentage load time overhead versus the static baseline, compared on a page-by-page basis.

that client-side code is cached at the client, and need not be downloaded. The bandwidth consumption overhead of the version with caching over the static one is minimal, around 6%. In the worst case, when server caching is unavailable, the extra work involved in retrieving keys and reconstructing the capability structure places this overhead at 23% over static[5].

In contrast with bandwidth requirements, the impact of using Micasa on page load times is more complex to characterize. We compared, pairwise, the total time needed to load all the comments and images on the page. The cumulative distribution of overheads in page load times over the static version is shown in Figure 2.3. For example, around 80% of all pages without caching have an overhead of 100% or less over static[6], and similarly, all pages with caching have no more than 40% overhead over static. Results shown are for 50 Flickr pages. For each page and version, we compare median load times over 21 repeated page views. Our sample set's static pages have 72.8 objects to retrieve on average (69.5 median), and have average load times of 1026.8 milliseconds (1006.5 median).

The overheads with respect to the static case are due to a number of factors.

[5]The median values are within 3% of the means, and we measured bandwidth consumption using Charles 3.6.5 HTTP proxy, with caching disabled in both the browser and proxy.

[6]$(t\_caching(\text{page}) - t\_static(\text{page}))/t\_static(\text{page})$

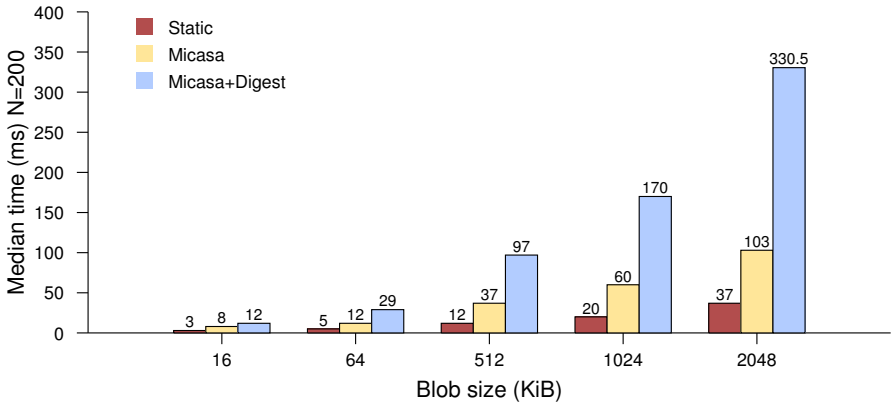

Figure 2.4: `getblob` overhead.

First and foremost, they are due to data dependencies that increase the overall time needed to construct pages. In the Micasa version without caching, four levels of capabilities need to be traversed before images and comments can be inserted in the DOM. Three of these levels are already unrolled with caching. Similarly, the caching version does not need to fetch keys and can skip signature verifications.

Generally speaking, capabilities on a same given level can be retrieved in parallel, but not before their parent list has been retrieved and parsed. In practice, this parallelism is subject to per-host-port connection limits, and a global limit set in the browser (16 and 35, respectively, in our experiment).

Second, in isolation, fetching blobs from our prototype CapServ incurs penalties over static file fetches from Apache 2.2.14. The overheads on the server consist of a base cost for capability verification (SHA1-HMAC), the Python language runtime, and a WSGI connector that passes data between the server application and Apache. This is added to the client overheads caused by the libeol invocation, and the client-checksum routine performed over response bodies. Figure 2.4 shows median overheads of performing `getblob` on various sizes over 200 trials, on a local network. The slowdown is approximately a multiplicative factor of 3 between static and Micasa, and 3 from Micasa to Micasa with client-side JavaScript SHA1 [92].

Third, as mentioned earlier, both Micasa versions benefit from local caching of client-side code.

Fourth, in our tested scenario, some Micasa pages gain an advantage over the static case due to increased data parallelism. In the Micasa cases, the browser can fetch data from up to ten stores based on the number of personal stores involved. Recall that in our static case, a single server delivers content. While we did this to model a startup service, more mature providers deploy multiple servers to improve page load times.

Overall, we find that with all factors combined, our Micasa prototype offers promising performance. Many further optimizations could be applied to improve its performance. For instance, our server could benefit from request batching, or recursive list fetches, and the client digest computation could be moved off the main thread into a worker.

2.6 Related Work

The idea of separating web application code from data is not new. It is a core concept in Web 2.0, and has been applied for many different purposes. Some use it to solve a form of data lock-in [15, 128], by empowering users to place data in personal stores in the cloud. Unfortunately, their storage models are mostly designed for single-user applications. In the case of unhosted, receiving updates from other users is only possible by granting coarse OAuth write access to a mediating service. They therefore do not address the challenges inherent to preserving social features without a central server, nor do they address building applications that are compatible both with and without the server.

W5 [34] also proposes an architecture allowing users to retain control of their data by separating it from application logic. W5, however, is designed to restrict data flow to providers. This falls outside the scope of Micasa; we focus on EOL functionality.

Menagerie [29] presents a system that allows applications to aggregate data across cloud services, by encapsulating access to objects inside capabilities. In both Menagerie and Micasa, capabilities provide uniform access mechanisms to objects stored on heterogeneous services, and are at the basis of sharing. However,

their goal is to aggregate data dispersed on the Internet. It does not allow service continuity without the provider. Our capability-based API alone provides features similar to other existing capability systems [74, 83]. Our system was developed independently, but it appears possible to create a portability layer to support our needed semantics on these other systems.

SPORC [25] and Frientegrity [26] address the problem of having groups of users collaborate on untrusted centralized servers. Whereas our goal is to test that application providers can provide continued application functionality, their main goal is to protect the user's data integrity by detecting misbehaving servers. They rely on developers using operational transformation for merging event conflicts, whereas we allow more familiar ways of programming. Our system's storage organization also resembles Persona [6]'s decentralized storage. They solve the particular problem of protecting data privacy in a distributed private social network context using attribute-based encryption, whereas we build a framework for letting applications disconnect from providers.

Many choose to do away with a centralized service and provide essential functionality inside smaller federated instances of the same application. Two famous examples are Diaspora [91] which replaces a centralized social network with federated "pods" users can join, and OpenPhoto [110], a photo-management and sharing application which allows users to run their own OpenPhoto servers. Users gain some control over their data privacy because they can choose the servers that will host it. However, their APIs are specific to a single application. Running multiple different applications with this design would require trusting and maintaining a different server each time. Our APIs on the other hand can generalize to multiple applications, and can benefit from the convenience of a central server.

2.7 Discussion and Future Work

As the Micasa framework adds complexity to an application, it is natural to ask what incentives developers have to use it. We believe the answer starts with users. If users are particularly interested in certain functionality, developers have an incentive to provide it. Micasa is motivated by the belief that many users would find features that mitigate against lock-in and EOL very appealing, and conversely, that

users find the lack of such mitigation a deterrent to investing their time and data.

In a Micasa application, the capital and operational costs of providing the service are effectively distributed across the service provider and selected storage providers (i.e., those hosting content for users registered for that service). Many new providers, and even some who run at scale already find it economical to leverage online storage in this fashion, for example, Netflix stores its entire movie library on S3. Furthermore, Micasa enables a new range of monetization strategies. For example, a service provider may have partnership agreements with select storage providers and flow user fees and ad revenue through these partnerships. After a user's storage provider is specified, an application could ensure that it be given a frame for serving an advertisement.

In our current prototype we require storage providers to export the CAPSI API. In our prototype, we demonstrate how to fulfill this requirement using an EC2 instance provisioned with suitable storage. Moreover, given that storage providers have an incentive to drive new business, if Micasa-style applications became popular, there would be an incentive to adopt the CAPSI API. Nonetheless, there is an obvious bootstrapping problem.

As part of our current efforts, we are working in two directions. First, we are exploring protocols for emulating the CAPSI API using the existing bucket and access control mechanisms on S3. We intend for the additional complexity on the client side to be encapsulated as part of the Micasa library. More generally, we assume that some diversity among distinct storage providers is to be expected and we are currently exploring a more flexible model. In this model, the library and monitor would support a number of storage APIs. Applications would interact with the "application-side" of the monitor using a generic application API. The "storage-side" of the monitor would be capable of interacting with any of a number of supported storage APIs. The monitor would translate between the application API and a particular storage API based on the domain of the client's storage provider. Even within this model, certain minimal requirements, beyond those currently fulfilled by current storage providers, may need to be fulfilled by any Micasa-compatible storage API in order to achieve both abuse prevention and proper access control.

Although we have not emphasized the following up until now, Micasa provides users with more control of their own content than in current centralized ap-

plications. In current apps, when a user deletes some content, she cannot be sure that well-crafted `get` commands by other parties will not retrieve the content. In contrast, in Micasa a user can revoke any or all of the capabilities for an object or delete the object on his store by virtue of his owner permissions. With respect to this added control we make three points. First, such added control may be appealing to users and used as a marketing tool by Micasa applications. Second, we emphasize that Micasa does not preclude an application provider from caching or mining user data in any form they like. In this sense, we do not believe Micasa deprives application providers from existing monetizing avenues, such as advertisement, or performance measures.

Finally, and conversely, the Micasa architecture may in fact provide the means for protecting the privacy of user data from the service provider. We are currently building cryptographic and key management APIs into the Micasa library to allow for confidential sharing among user defined groups. As future work we will explore practical techniques for the Micasa monitor to police against exfiltration of content and keys to the provider. Such privacy-preserving apps could still allow for provider-side caching of encrypted objects. Services could still be supported with untargeted ads. Users may be willing to directly pay the incremental costs between targeted and less targeted ads for the added privacy. Whether they are willing is a market question, one which our architecture allows developers to explore.

2.8 Conclusion

In this paper we introduce the Micasa architecture for writing applications which treat service provider EOL as a recoverable failure mode. The model assumes an ecosystem of storage providers that export a common API and a client-library for writing applications that adhere to this API. Micasa users cache client-side application code and upload their content to personal stores rather than to the service provider directly. While the service provider is in business, Micasa applications enjoy the benefits of existing centralized services, but if the provider discontinues service for any reason, Micasa clients can continue to use the application in an unplugged mode. When unplugged, Micasa clients retain the functionality and data access promised by the developer in perpetuity. Micasa also enables users to audit

the developers adherence to the expected unplugged behavior at any time.

We demonstrated the feasibility of the architecture by building two applications, each with different data sharing characteristics. In addition, we emulated the load times that Flickr pages would have if delivered as a Micasa application, and found the overhead of the numerous `HTTP` fetches acceptable due to the concurrency afforded by modern web browsers and the possibility of future speed improvements for client-side computations. Hence, a provider building applications with the Micasa platform can offer minimal EOL risk to new users while still being able to deliver features and performance comparable with existing centralized services.

Chapter 3

Beeswax: Preventing Data Exfiltration in Private Web Apps

3.1 Motivation and Overview

Recently, users are showing increased reticence in giving up their privacy, enough for some service providers to market their privacy-friendly data policies. For example, ChatStep [84], niltalk [107], Google Hangout's "off-the-record" conversations [103], and Slack [122], all make various claims that the data that users pass through their service will not be stored or will be stored but not monetized. Since such claims are enforced on the server side, the only assurance that users have is the knowledge that providers have a strong financial incentive to avoid a public breach of these claims. On the surface, web applications like noteshred [108] (encrypted notes and messages), Subrosa [123] (encrypted chat), and Cryptocat [88, 89][1] (encrypted chat) provide more assurance of confidentiality as they use client-side cryptography. However, after having been lured to an application by promises of privacy, how would a user know that the client-side code was not exfiltrating private plaintext or keying material?

We posit here that any application that implements end-to-end cryptography

[1]Cryptocat was initially written as a web application, rewritten as a browser extension around 2012, and then rewritten for desktop in 2016. Forks of earlier versions are available online, one of them being Cryptodog.

must be considered by the user as being part of his/her trusted computing base: even if the code itself is not intentionally exfiltrating private data, any vulnerability in the code might be leveraged to extract private data. Ideally, the market would offer a large number and variety of privacy preserving applications: messaging apps, photo sharing apps, full-featured social networking apps, webmail clients, etc. If a consumer would like to benefit from the functionality of a handful of applications, he will have to implicitly put all of them in his TCB. To make the issue of trust even more problematic, many important safeguards (such as checking the fingerprint of an application version that has had a thorough open-source review) do not fit with the deployment model of modern web applications as HTML, CSS, and JavaScript are modified frequently and pages often contain dynamic content.

3.1.1 A Platform Approach

In current private web applications, it is all or nothing: either you don't use the application or there is no isolation of critical code and data. Because of this, every new application adds its entire code base to the TCB. The approach we take in this paper is based on the principle of least privilege: isolate highly privileged code and sensitive data and export a narrow interface to this code. We observe that once this is done, multiple applications can use the same sensitive code base via the APIs without adding to the TCB. We propose a platform approach to building confidential web applications. The goal of our approach is twofold: 1.) to allow for applications that have rich interactions with service-side functionality, and 2.) to reduce the assurance of an ecosystem of such web services to the assurance of the platform code.

In our proposed approach, the responsibility for the overall functionality of a service is split between our client-side platform, Beeswax, and the service provider's code (both client- and server-side). The platform is responsible for, among other things, managing and isolating keying material and for performing standard cryptographic operations on behalf of the application through an API. The application is responsible for the rest of the functionality, including its look and feel, sharing, and distribution of (encrypted) content. For example, the application is responsible for designating DOM elements as private, e.g., after user action. The platform takes

care of accepting user input to a private DOM element and shielding that input from the application. The platform similarly displays data in private elements to the user but shields it from the application. The platform is responsible for providing an unspoofable indication to the user about whether a DOM element is private, and if so, the identities of other users with access to the content of the element.

As a result of this split, the technical community can focus its scrutiny regarding whether a given set of privacy properties are properly implemented onto a single code base. The intent is that focusing scrutiny on a single code base, as opposed to spreading the scrutiny across a large number of applications, will generate greater assurance in the claimed privacy and security properties, e.g., the impossibility of exfiltration of private user data or keying material to the application.

3.1.2 Adversarial Apps and Other Threats

Our goal is to ensure that users need not place applications that use our platform in their TCB. However, by placing applications outside the TCB we must assume them to be malicious. While it may seem counter-intuitive to assume that an application dedicated to preserving privacy should be considered malicious, it would not be unthinkable for a service to lure users into using an application with promises of privacy via the use of a privacy platform, only to subvert the platform to retrieve private data for commercial or nefarious gains.

Thus, the platform must provide privacy guarantees even in the face of an application purposefully trying to circumvent its defensive mechanisms. We assume that an application may attempt to: 1.) exploit a vulnerability in the platform code in order to retrieve data in UI elements designated as private or to extract keying material in isolated storage or 2.) trick the user into entering private data into a UI element that the application has not designated to the platform as being private.

A top level goal is to maintain the integrity of the platform code (e.g., the mechanisms that isolate data in private UI elements from the application) even in the face of attacks against the platform. We discuss our software integrity defenses in detail in Section 3.3.2. We cannot, of course, guarantee code integrity completely. However, as discussed above, we believe the benefit of a platform approach is to focus the scrutiny of the community on the code base of the platform to maximize the

chances of detecting and fixing vulnerabilities.

We assume that the application may attempt to mislead the user about the read/write permissions of a given UI widget via visual trickery, e.g., transparent overlays or rapid changes of focus. We take it as a given that users cannot achieve even the slightest of privacy guarantees without paying at least some attention. The goal of our platform is to provide defenses against active UI attacks assuming only moderate user attention. For example, Beeswax provides an indication to the user in the browser's toolbar about whether the UI element with the current focus is private, and if so, what other users have access to the data in that element. This is detailed in Section 3.3.3. Short of extremely sophisticated per-user behavioral modeling, as long as some users are sufficiently motivated to protect their privacy from UI spoofing, an application that occasionally mounts such attacks would be identified by the community.

Our platform exchanges messages over the network, via the application and a pub/sub service such as Twitter. Beeswax must thus also protect the confidentiality and authenticity of messages in flight. Beeswax uses well-known cryptographic protocols for such purposes. We do not defend against denial-of-service, but failure to deliver messages would only result in poor user experience, without negatively affecting security. Our platform is immediately deployable, but relies on the PKI trust rooted in the browser for HTTPS communications.

To bootstrap trust in the messaging, we assume users of the platform are also registered users of a pub/sub service. For this paper we use Twitter but other services could also be used. We assume that each user, via out-of-band mechanisms, makes a personal determination of the validity of the binding between the pub/sub (Twitter) account name and a person. We assume that when choosing to share, a user will only use account names that meet some personal threshold of trust. For example, an account name may be considered trusted when it has been shared via a number of channels and the stream of updates is completely consistent with what one presumes to know about the person supposedly bound to the account name. We cannot protect users from spoofed accounts if they put no effort into ascertaining the identities of account holders.

We do not combat attacks on the underlying OS or browser. We consider them as part the platform's TCB: any security improvements to them will be inherited

by our platform. In any system implementing end-to-end security, the privacy of a user at one end may be compromised by a breach at the other end. Users of Beeswax must therefore trust the integrity of the systems of the users with whom they interact.

3.1.3 Beeswax

In this paper we present Beeswax, a platform for developing private web applications. In designing Beeswax, our goal is a platform that:

1. supports the development of rich interactive applications with custom look and feel,
2. provides transparent key management,
3. enables applications that give users control of the permissions of UI elements that the application has labeled as private.

Assuming that selected Twitter identities are trusted, and that the cryptographic libraries we use are implemented correctly and computationally secure, we claim the following, in parallel with the goals. Beeswax:

1. adds only minimal complexity and lines of code to an application and incurs minimal overhead,
2. prevents exfiltration of keying material, and spoofing of identities by an application,

3.a prevents an attentive user from reading (writing) private data from (into) a non-private UI element by providing an unspoofable indicator for the permissions of the UI element in focus,

3.b prevents data entered into a private UI element from being leaked to the application or unauthorized users.

To validate the first claim we build two applications: a privacy-enhanced version of the IRC web client Kiwi IRC [105] that allows encrypted messaging and

an encrypted photo-sharing gallery called PicSure built from the ground up. We report on their construction and platform overheads in Section 3.4.

We cover the isolation of keying material with our platform's API in Section 3.2.1. We argue for the correctness of our identity and key management in Section 3.2.2. Isolating plaintext from the application (and unauthorized users) is covered by the implementation, Section 3.3.1 and Section 3.3.2. Finally, we describe our defenses against UI-redressing attacks in Section 3.3.3.

3.1.4 Overlay Versus Platform

An approach related to our own is embodied by the browser extensions Priv.ly [113] and ShadowCrypt [30], which we call overlays. Overlays do not require the cooperation of the web service to which they are applied and so work with existing services. Their aims are similar: a user may protect text by declaring input areas as private. Private posts are submitted as encrypted markers (inline, or external links) which are exposed as plain text by overlaying isolated elements. In its current implementation, ShadowCrypt does not meet this isolation goal. In Section 3.6 we show it has many vulnerabilities that allow us to craft attacks that retrieve plaintext.

While we believe that it is possible in principle to build an overlay that properly isolates keying material and user content using techniques similar to our own, the overlay approach, in general, leaves unsolved two significant impediments to security and deployment: UI spoofing and key management.

UI SPOOFING. ShadowCrypt [30] explicitly excludes UI spoofing from its threat model. Indeed, without hooks into the application state, it is difficult, if not impossible, for a security layer to provide an unalterable indication to the user about the privacy characteristics of a UI element. For instance, the use of coloured borders and floating padlock icons around confidential elements in a page (as used in ShadowCrypt) are not sufficient to protect against malicious UI spoofing.

KEY MANAGEMENT. Overlays remain entirely separate from applications, requiring that their key management be done out-of-band of the application. In practice this means that users of an overlay must, for example, email or text a symmetric key for each application stream and friend. Such *ad hoc* key management severely

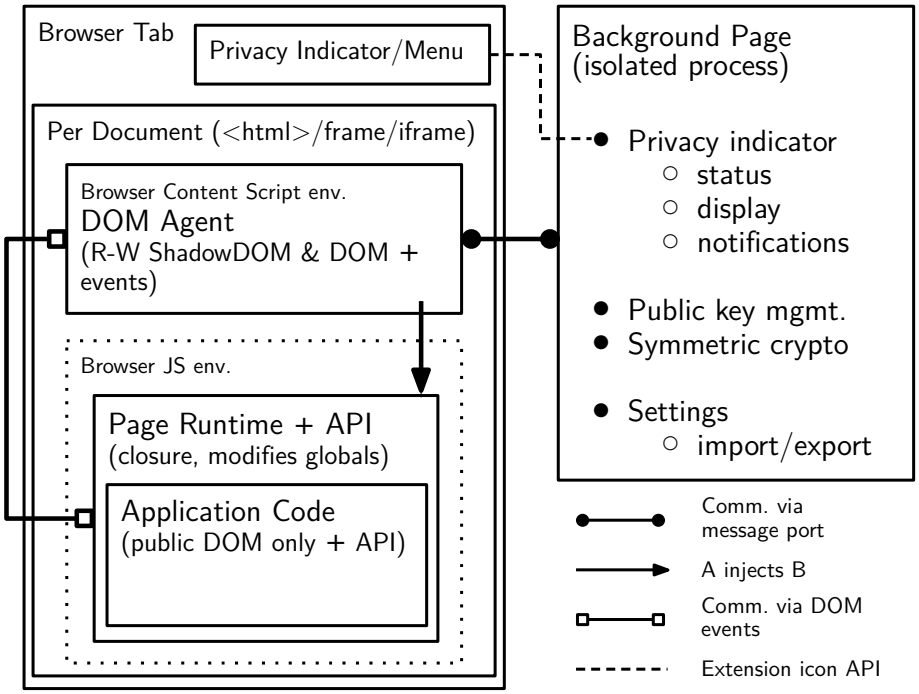

Figure 3.1: Division of the Beeswax platform into components.

limits adoption of an overlay to all but a few motivated users. Our platform approach, on the other hand, allows keys to be exchanged securely in-band of the application.

In short, a plethora of stand-alone privacy applications are difficult to trust. An overlay will break some functionality, is susceptible to active UI attacks, and requires unrealistic user effort for key management. We posit that tight cooperation between developers and a security platform is necessary to achieve an ecosystem of easy-to-use applications that provide strong assurance of privacy.

3.2 Architecture

Beeswax is built as a Chrome browser extension. It therefore ties together multiple extension components, as per Figure 3.1. Below, we briefly describe these components and our usage of them in Beeswax. We also rely on another browser technol-

ogy called ShadowDOM, which we also briefly describe below. We mostly mention in this paper the security properties of Chrome extensions and ShadowDOM that are of high relevance, but they are otherwise well documented [7, 14, 133].

BACKGROUND PAGE. A Chrome extension consists, in part, of a background page: a JavaScript program that the browser isolates from other pages and extensions. The browser provides the background page with its own isolated local storage and with access to several privileged browser APIs. A background page may also include HTML to be run in a tab. Beeswax relies on the background page's isolated context and its ability to run continuously for accomplishing a number of crucial tasks. These tasks include:

- public key management: identity binding & validation, public key store, and key agreement;
- symmetric key cryptography: key generation & indexing, and cryptographic computations;
- control of the privacy indicator (described below): status, display, and user notifications.

DOM AGENT. An extension may include a content script. Every time a page is loaded, the browser loads the content script in a separate JavaScript environment that is paired with the JavaScript environment of the page. The pair share access to the same DOM, but none of each other's JavaScript variables or functions [87]. We call the Beeswax content script the DOM Agent.

PAGE RUNTIME. By virtue of having access to the DOM, the DOM Agent can insert additional scripts in the page. These injected scripts share the same object space as the page. Extensions can be configured so that the browser runs the content script *before* the page is loaded. Beeswax uses this early-load capability to install our platform's API (covered in Section 3.2.1) and to create a modified runtime for the page which isolates the API implementation from the page code.

SHADOWDOM. Another browser mechanism useful for our purposes is the recent ShadowDOM. The technology allows DOM subtrees (called "shadow trees") to be grafted to a host DOM element and override its rendering: the browser renders the

contents of the shadow tree in lieu of its host. The technology became officially available in Chrome version 35 beta (May 2014). While the technology alone does not allow full isolation between a host and its shadow tree, mechanisms in Beeswax repurpose shadow trees to render private data in plaintext, but present it as ciphertext to the hosting application.

PRIVATE AREAS. When an application designates a DOM element to host a private area, the background page, content script, injected script, and ShadowDOM mechanism work together to deny the application read and write access to its ShadowDOM. The injected script provides the APIs for an application to create a private area. When this API is invoked, the DOM Agent creates a shadow tree rooted at the chosen host element. User data is written inside the shadow tree and will be rendered by the browser. The injected script and DOM Agent relay user data to the background page and back for encryption and decryption. An application cannot read the shadow tree directly, but can extract ciphertext from a private area using the API. It can then embed the ciphertext into its various communication streams.

PRIVACY INDICATOR. For each extension, Chrome reserves an area on the right side of the toolbar for drawing one icon. This icon can be drawn dynamically and can adorn a small amount of text called a "badge". The extension icons are unspoofable: a webpage cannot render UI elements outside the browser window, so it cannot cover up an extension icon. Similarly, an application could launch a popup window to redefine its boundaries, but the browser ensures that popup windows are always rendered on-screen and with an address bar. Moreover, the browser ensures address bars in popups cannot emulate the look of a browser tab toolbar.

A malicious application might trick a user into inputting private data into areas that are not private. We employ the Beeswax extension icon as a privacy indicator to defeat UI-redressing attacks. The DOM Agent informs the background page of references to DOM elements having received the last user event. The background page uses those references to update the icon and badge appropriately. Furthermore, if the current UI element is part of a private area, the icon can be clicked to reveal the user names of those with access to the data.

Table 3.1: A subset of the Beeswax Page Runtime API.

Category	Methods
Friends	`get_friend(userid)` → `fr_chan`

Streams	`new_stream()` → `streamid`
	`invite(fr_chan, streamid)` → `invitation`
	`accept_invite(invitation)` → `streamid`

Private UI	`make_priv(dom_el, streamid)` → `bool`
	`is_priv(dom_el)` → `streamid | nil`
	`get_cipher(dom_el)` → `cipher`
	`put_plain(dom_el, cipher)` → `bool`

3.2.1 Beeswax Operation and API

We present the operation of Beeswax and the APIs, listed in Table 3.1, that invoke those operations. The table omits appearance modifier calls (introduced below) and one call to enable the extension in the page. Our platform uses three standard asymmetric and symmetric cryptographic methods: trusted bindings between public keys and identities, authenticated key agreement using asymmetric cryptography, and communication channels protected by symmetric key cryptography. By design, the complexity of these operations is hidden from the application: the API is concerned with opaque handles to keys, rather than keying material.

IDENTITIES AND PUBLIC KEY DISTRIBUTION. We could not find an existing public key binding and distribution system that fit easily with our deployment model. We sought to use Keybase.io [104], a promising approach to distribute keys across multiple social networks. However, it does not currently support multiple public keys per user. Moreover, formal documentation for their proofs of identity is lacking.

Another alternative is PGP-based tools. PGP's trust model is difficult to make fully programmatic, except with complex policy configuration. For example, even when a key with a given ID (e.g., an email account name) is pushed into the PGP key base by one party, it is trivial for another party to push another key into the key base with the same ID. In principle these can be differentiated by web-of-trust

signatures by other parties. But for most users, deciding which public keys to trust would need to be done on an *ad hoc* basis depending on the web-of-trust of each particular key/ID pair.

For Beeswax, we developed a key distribution mechanism which uses a pub/sub service both for identity and key distribution. We use Twitter, but other services would be suitable as well. Firstly, for a pub/sub service to be usable by Beeswax, it should be possible for Beeswax users to verify a binding between a person and their account ID on the service. For instance, the content on a Twitter feed should match what one would expect the feed's owner to post. This inferral can naturally be avoided if one already knows the account ID of the interlocutor. Secondly, the pub/sub service should authorize its users so that only the owner of a feed can post to it. This way, when certificates are retrieved from a feed, we can expect them to have been posted by the feed's owner. Thirdly, subscription to a feed must be done over an authenticated channel. Lastly, it should be suited to store certificate information, possibly using fragmentation or image steganography to bypass length or format limitations. The roles of identity provider and key distribution can be dissociated, but if and only if users on the key distribution service are authenticated using identities from the identity provider (e.g., distributing certificates with Twitter IDs on source code repositories would hypothetically work if the repositories could authenticate using Twitter accounts).

Initially, each user creates a self-signed certificate (cert) embedding his/her Twitter ID. Users then publish these certs over their Twitter feeds. Users then pull certs from their own feeds and the feeds of others with whom they wish to communicate, and sanity-check these certs. Beeswax does not support anonymous keys – it provides confidentiality, but anonymity is beyond its objectives.

Normally, self-signed certs have no identity integrity on their own. The key idea here is for the identity in the cert to be the same as the name of the authenticated (i.e., HTTPS) channel over which the cert was obtained. The extension periodically compares certs installed locally to those on the key distribution service, and raises a flag if they differ. In this way, Twitter IDs and their associated authenticated pub/sub channels become roots of trust. We assume that users decide for themselves, based on out-of-band evidence, which Twitter username/identity pairs are sufficiently trusted for the purposes of a particular interaction. We analyze the

security of this key distribution scheme in Section 3.2.2.

Applications must obtain Twitter usernames at registration to use Beeswax. As we will show, a user who registers a Twitter account to which it does not have access will not succeed in completing key agreement protocols with other users of that application. We support configuring multiple Twitter accounts, but only one may be used at a time. For simplicity, this discussion assumes the extension has been configured with a single account.

Beeswax will make reads and writes to a user's Twitter account on behalf of the user. To do so, the user must first register his/her Twitter account name in the Beeswax settings. Subsequently, whenever the user is logged into his/her Twitter account, Beeswax will have the credentials necessary to post to the user's account.

To proceed, a user must install asymmetric keys into Beeswax to bind to the installed Twitter username. Beeswax provides two mechanisms for this. In the background page tab, Beeswax can generate an encryption key pair and a signature key pair and install them into the public key database using its `gen_pub_keys` routine. Beeswax can also import public keys the user owns with its `import_keys` routine.

Once the user's Twitter ID and public keys have been successfully installed, the background page invokes the `distribute` background daemon. This process periodically reposts self-signed certs of a user's public key pairs to the user's Twitter account. This daemon also periodically polls the user's feed for posts that claim to be self-signed certs of the user. If a cert is invalid, its public keys are not those installed into Beeswax, or its embedded Twitter ID cert is not the user's, a flag is raised. Details of this process, such as expiration and revocation, are described in Section 3.3.4.

The Beeswax background page also has a background process called `validate` that periodically fetches the recent posts of other users. When `validate` is first called, it scans a user's feed for the most recent self-signed cert. If the cert is valid, and the username in the cert matches the Twitter account name, it installs the username and public keys in the public key database, otherwise it raises a warning. On subsequent calls, `validate` tests for the validity of the certs, checks whether the embedded Twitter ID matches the feed, and checks whether the keys are the same as those already installed. If not, it raises a flag.

The background page operates as soon as, and as long as, at least one browser window is open. Periodic tasks are rescheduled immediately when the browser starts up or when connectivity is resumed. Operating 24/7 ideally minimizes problem detection delays, but is not strictly necessary.

KEY AGREEMENT AND FRIENDSHIP CHANNELS. In order for a communication stream between two parties to be identified as private via the Privacy Indicator, an application must first enable a friendship channel, i.e., a secure bi-directional signalling channel between two parties. When a user selects a friend (identified, in part, by a Twitter username) inside an application, the application makes a call to `get_friend`. If the potential friend's public key is not yet in the public key database, the background page makes the `validate` call described earlier. Once a key is installed, the background page, with the help of the communication channels of the application, performs a key agreement protocol (Section 3.3.4). The net result of a successful protocol is a set of symmetric keys that enable secure communication between the two parties. If the key agreement protocol runs to completion, `get_friend` returns a handle to the keys for the secure friendship channel. This channel is used subsequently by friends for passing reference values, metadata, and symmetric keys for collections of private data that we call streams.

STREAM. A stream is a series of media messages between one, two, or more users. The function `new_stream` creates new streams. During this call, Beeswax generates one symmetric AES key which will be used in authenticated encrypted mode to encrypt all messages from the stream. The creation operation returns the stream identifier `streamid` (a key handle) so that it may be stored by the application and retrieved subsequently. The user creating a stream is said to be that stream's *owner*.

SHARING. Sharing in an application is achieved by distributing stream handles. The owner of a stream can use the `invite` method with two key handles: one designates a friendship channel and the other one designates a stream. The operation is asynchronous, which allows the platform to ask for user permission (e.g., "The application is inviting `@Carl` to a stream in which `[@Alice, @Bob]` already take part, do you accept?"). The call generates an invitation message over that

friendship channel. One invitation is sent over each friendship channel between the owner of the stream and its participating users. That is, the friendship channels are one-to-one, but multiple friends can be invited to the same stream.

The invitation messages are authenticated and contain the encrypted stream key, the stream ID, and metadata indicating their provenance. On the invitee's computer, calling `accept_invite` will reconstruct the stream handle. The provenance allows our UI to determine and list a stream's participants.

PRIVATE UI. Once streams have been created, the application may choose elements of the page to host stream content for display. First, `make_priv` transforms a regular DOM element (`dom_el`) to "host" private content. The platform clones the host's subtree into a new private area and inserts the area as the children of a shadow tree on the host – which hides the original subtree from view. The platform associates the host element with that stream for the rest of the element's lifetime. The application keeps references to the host element (it is part of the regular DOM), but our mechanisms prevent the application from querying the DOM to access the private content. To query whether an element hosts private content or not (i.e., if it has been passed to `make_priv` previously), we provide `is_priv` as a convenience.

To read the user's encrypted input, the application calls `get_cipher` with the host element `dom_el`, which will cause the extension to read the contents of elements, and encrypt them with the associated stream key. To display it, ciphertext information is passed back into `put_plain`, during which the extension replaces the visible contents of the element with those of the decrypted message.

Beeswax supports secure input and output (display) of both text and images. The element passed to `mark_priv` determines a private area's type: e.g., `<input>`, `<p>` host text and `<img>`, images. To provide the application more control over the private area content, which it cannot access directly, the platform ships with "appearance modifiers", functions part of our TCB (omitted from Table 3.1), that the application can call to change an area's look and feel. We list limitations in Section 3.5.

Beeswax fully controls the flow and contents of DOM events falling into (and outside of) any private area. Our privacy indicator monitors these events: at any given time, it can confine events to only certain private areas, and block others.

This interposing of events protects users against UI-spoofing.

3.2.2 Security of Beeswax Key Distribution

We distribute keys over the pub/sub service, Twitter. For this discussion, we assume that if a Twitter account is hacked and the true owner loses access to the account, the account can be disabled by contacting support. Such a scenario is outside of our threat model.

UNAUTHORIZED POSTS. Consider a malicious party other than Twitter. Such a party may craft key pairs for which it knows the private keys and attempt to post the cert for those keys to a user's Twitter feed. If successful, several other users may install the malicious public key into their Beeswax keyring for the victim. We note that such an attack can only be successful if three things are true: the party gains access to the victim's Twitter account, the victim does not notice the bogus cert in its twitter stream, and the malicious party has gained access to the victim's Beeswax extension and installed its bogus key pair. The latter is necessary because the distribution protocol requires the background page of a user to ensure the user's published certs and those installed locally have matching public keys. We consider it unlikely that all three conditions above will be true.

Although Twitter itself has write access to every Twitter account, it is not in a position to install key pairs in Beeswax extensions. Thus, Twitter cannot simply try to post a crafted cert to a user's feed without the user noticing. However, Twitter could perform a fork attack on a user's account. That is, it could present the correct replay of a user's posts to the user, but present an incorrect feed of posts to the user's followers. This fork could trick that user's followers into binding a bogus public key to the victim's Twitter ID, a public key for which Twitter had the corresponding private key.

We do not attempt to provide protection against such a strategy by Twitter. Nonetheless, we posit that maintaining fork-consistency after having lied (i.e., serve veridical tweets to the key owner, but lie to all others) would be prohibitive. Amongst applications in which the target user and friends were already enrolled, and Twitter's indisposition to man-in-the-middle, some evidence of the subterfuge would surface.

TRUSTING STRANGERS. A malicious party may attempt to register a Twitter handle designed to deceive other users about the identity of the account's owner. A user who asks her extension to subscribe for key broadcasts on a given Twitter ID should perform due diligence, up to her desired level of assurance, on the account owner's true identity (e.g., using out-of-band channels). Our scheme does not attempt to protect unvigilant users.

LIMITATIONS. Twitter does not guarantee that tweets will be searchable beyond about a week after their posting. Not all tweets are indexed in the same way, and what determines that a tweet will remain searchable past this window is not made clear [126]. This API behavior forces extensions to continually re-publish their identities to prove possession of the signing private key and permit discovery by other users of Beeswax applications. This API limitation, in fact, encourages good practice. It is for this reason that we expect a working validity period to be just shy of a week.

3.3 Implementation

Our prototype Beeswax platform is implemented against a stable version of Google Chrome (initially version 40.0.2214.95) and can be installed in only a few clicks. This allows our work to be immediately deployable on the Web. It is composed of about 5000 lines of code, including comments and HTML, and excluding third-party library code. The code for our prototype and applications is open-source and is available online on our project page [79].

3.3.1 Secure Display of Private Data

An intricate aspect of the implementation follows from the need for private areas to be hidden from the application, but visible to the user. It also aims to allow the application to preserve its desired look and feel.

CREATING A PRIVATE AREA. Our Page Runtime modifies the application's environment to prevent Beeswax-enabled applications from creating new shadow DOMs and retrieving the root of existing ones. As a result, transforming host elements into private areas can only be done via the content script, with the use of

the `make_priv` API call. We support two types of private areas: generic text and images, but this could be extended to other media (canvas, file attachments, etc.).

We call HTML5's `Element.createShadowRoot()` to create a ShadowDOM subtree. The created shadow root is made available in a single getter property called `elt.shadowRoot`, and that property only. Nullifying that property suffices to hide the private area from the application DOM. The application cannot recover nodes from shadow trees by traversal nor with typical selectors (e.g., `document.querySelector`). However, the shadow trees remain accessible to the content script.

Browsers render what is called the "composed" DOM tree, a stitched-up global view of the main DOM tree and shadow roots. Mangling the `shadowRoot` property of nodes in the application environment does not affect this composition, as the browser still uses internal references to form the correct tree. The composition rules are such that the content of the shadow root (which will contain the private data) is rendered in place of the content of the host node in the composed tree. The rules are complicated by the nesting and relocation of multiple roots, but we eliminate this concern by preventing applications from creating new roots outside `make_priv`.

TRAPPING EVENTS. Due to the way events propagate in the DOM, events targeting private elements may leak private information, such as keycodes, selection of text in the page, or data input changes, into the application's event handlers. One challenge of our implementation is to prevent leaks without breaking the event system applications depend upon. The platform traps events that carry sensitive information. Not all event types can target private elements, so we apply a first round of filtering to move certain events back on the fast path. For instance, events that only target the `window` or `document` (e.g., `DOMContentLoaded`) propagate unmodified.

To ensure our filters are effective, we arrange for the platform to inspect every event dispatched *ahead* of the application. We rely on two properties enforced in the browser: first, that the order in which event listeners are registered is honored during dispatch, and second, that event listeners cannot be removed without a reference to the identical object given at registration. The rules of event dispatch are intricate and are omitted here for brevity. We make use of the former property when

our Page Runtime is injected before the application, and we make use of the latter by hiding the Runtime's event listeners in a closure, thus preventing the application from de-registering our listeners. Although the order of invocation of listeners on a given object (e.g., `EventTarget`) is unspecified by DOM Level 2 standards, it is predictable within a particular browser. This is rectified in the newer DOM Level 3 [131], where orders of invocation and registration of listeners must match.

Filtering only the event types for which the application has handlers registered would be most efficient, but is hard in practice. It is straightforward to obtain a complete view of DOM Level 2 listener (de-)registration because it is centered around a single prototype method (i.e., `addEventListener`) that can be interposed. However, legacy DOM Level 0 events can be registered independently on all objects, either inline in the HTML or with a script (e.g., `img.onload = my_func;`). We are not aware of ways to disable this older event model, and intercepting all accesses to all `on*` properties of all objects (and along their prototype chain) is impractical. Our Page Runtime remains conservative, pre-registering all event types for which there exists an `on*` method before the application does (around 100 different types of events are registered this way, e.g., `onfocus`, `onblur`, etc.).

SANITIZING EVENTS. Our sanitizer functions will erase properties that might reveal information about the contents of the private area, such as keycodes or selection information. When an event is dispatched on a node inside a shadow tree, a private area in our case, standard event dispatch undergoes a process called event retargeting. In this process, the effective target of an event is updated when it crosses a shadow tree boundary. This is so that the target of an event (as in `event.target`) always points to a node within the same subtree as the node on which the current handler is installed. This retargeting process is in line with ShadowDOM's original intent, i.e., to hide the complexity of a sub-layout from the parent layout. Consequently, if our `window` filter detects an event targeting (or relating to) the host element of a private area, the actual node targeted is either the host element itself or an element within the private area. In either case, the event is flagged for sanitization.

The platform is configured with a static list of properties that should be erased for the various event types. Depending on the nature of the property to hide, we

either use `delete` to remove the property from the event object or rely on non-reconfigurable getters (that return a constant instead of the true value) along the prototype chain. During event propagation, each event listener will receive the same event object, so our modifications persist across the event's entire lifetime.

We considered it appropriate to only erase the sensitive content from the event object, rather than stopping its propagation altogether. This is useful, for instance, to determine if a user has started typing. In the future, we could also base these decisions, i.e., to sanitize or stop, on a configurable application policy. We could also delay the events, for instance to prevent a timing attack that would infer keystrokes, but this is outside the scope of this paper.

REVEALING PRIVATE CONTENT. We assume the application running in the page cannot simulate keypresses on behalf of the user. This ensures all private data entered comes from users. Applications can indeed synthesize UI events, but in Chrome we can differentiate those from true user events. The system clipboard could constitute an exfiltration path, but modern browsers cannot read its contents without the user's permission. The page could access regions of the page that are selected, i.e., with `window.getSelection()`), but those can be interposed. HTML5 comes with a Web Clipboard API with a programmatic RW capability, but it cannot amass private content without user action.

Styling rules in the page can affect the look of private areas, and may shift content around. While we are not currently aware of attacks that would infer precise contents of a box by observing its response to repeated resizing, such an attack is conceivable (e.g., by prescribing crafted fonts, or forcing text to wrap in different ways). A related attack exists where links are styled differently based on their "visited" status (a CSS `a:visited` property exploit [90]), but this has since been fixed by limiting the capabilities of CSS in certain selectors. We imagine similar patches could be applied if other such side-channels were to surface. Another remedy could be to impose fixed dimensions on private areas.

3.3.2 Integrity of the Page Runtime

The security of the platform, and the safety of the user's private data, hinges on the inability of the application to modify the Page Runtime's behavior. If application

code were to compromise the Runtime, it could exfiltrate private content. The Runtime constitutes a self-protecting reference monitor to the privileged objects and functions it defines. It is a *lightweight* modification to the browser (as defined in previous work on self-protecting JavaScript [54]): it is trusted and does not parse or modify the code of the application. To the application, the Page Runtime appears as a built-in browser API. We use language-based techniques, such as closures and function wrappers, to reach the following goals of protection:

1. Globals, methods, and object properties used in the Runtime are those original implementations provided by the browser.
2. The application cannot tamper with objects and functions of the Runtime.
3. Functions wrapped by the Runtime do not reveal the unwrapped implementation to the application.

We have not directly used tools from previous work [42, 43] to generate code for our Runtime, but our construction defends against the tampering attacks they describe: prototype poisoning (when functions, getters, and setters are rewritten by the application), abusing the callchain (e.g., reflection over `arguments.caller`), unsafe accesses on application-provided objects (safeguards against objects which can change appearance between time-of-check and time-of-use), unsafe casts (e.g., implicit string casts), etc. We explain how we achieve the first goal next. Goals 2 and 3 are handled with the same care, but we omit their descriptions for brevity.

First, before the application starts, a (pristine) copy of each needed global is passed as an argument to an all-enclosing anonymous function – in other words, we save global objects and functions into local scopes. These locals are used instead of accesses via the `window` object. Verifying that only symbols local to the Runtime are referenced is tedious, but this task could be automated with a static analysis tool designed for this purpose.

Second, accessing properties with the dot operator or brackets, e.g., `x.y` or `x["y"]`, might invoke getters and setters that have been defined by the application. Where needed, we rely on built-in object reflection mechanisms to read application objects, for instance with `Object.getOwnPropertyDescriptor`. In cases where the Runtime must invoke a prototype method on an application object, e.g.,

`obj.hasOwnProperty('x')`, we avoid the dot completely by first wrapping the method's `apply`:

```
function wrap(m){ return m.apply.bind(m); }
var HOP = wrap(Object.prototype.hasOwnProperty);
```

Assuming `HOP` is saved to a local in the Runtime, the example can then be safely re-written as `HOP(obj, ['x'])`. This is a refinement over the approach presented in [43].

Third, access to still-undefined object properties in the Runtime cause a lookup in the prototype chain, which could find a hit on a property of `Object.prototype`. For instance, if the application were to install a setter property called `key` on `Object.prototype`, statements in the Runtime code as seemingly innocuous as `var data = {}; data.key = "X";` would leak the value to the setter. To eliminate this possibility, and allow base objects to be used inside our code with peace-of-mind, we *freeze* `Object.prototype`. Prototype lookups on objects could be avoided by other means, albeit tediously. Literal object definitions, for instance, do not cause the setters to be invoked in the prototype chain. In the same example, `var data = {key: "X"};` would *not* invoke the setter.

Using `freeze` is simpler than breaking prototype chains [43] and eased development. Freezing `Object.prototype` is convenient, but we have found it to break some websites (Gmail) and some libraries (d3.js) in subtle ways, so we also had to allow certain redefinitions, such as `toString`, to restore expected behavior.

Future language features could allow discovering hidden content (e.g., a new exception mechanism that allows inspecting stack variables, a new unhandled event type, or additional object introspection) and could affect the integrity of the platform. We cannot claim our defenses are future-proof. On the other hand, the platform approach allows code to be exposed and vetted by enthusiasts and professionals. The software platforms underlying the Web evolve constantly – everything built on top of them must necessarily co-evolve, or die. We do not believe future uncertainty should be a deterrent to the creation of novel privacy-protecting technologies.

We note that the number of global wrappers introduced by the Runtime is tractable: only for event listener registration, selected event getters, and `shadowRoot` interactions (all discussed in Section 3.3.1). We have been diligent in consulting

specifications and available documentation to cover possible aliases and equivalent functions. Also, because it is difficult to determine, without inspecting the browser's source code, whether a built-in (e.g., `appendChild`) internally accesses object properties that can be modified by the application, we perform DOM manipulations in the content script rather than in the page. Hopefully, further review of the published code would clear out any oversight on our part.

3.3.3 Privacy Indicator

Ideally, we would like to have visible at all times the full list of participants involved in a particular stream. Unfortunately, Chrome allocates space for no more than a single icon per-extension in its decorations. The icon can display a badge of text, but even that is limited to 4 characters. The compromise we found was to use the icon as a notification and use its popup activation menu to display additional details.

The indicator toggles between two modes: "protected" and "unprotected". As long as there is no user activity, the indicator rests in its default "unprotected" state. However, when the Agent receives a keyboard or click event for a private area of the page, it transitions the indicator to "protected mode" for *that* area. The indicator changes its appearance to notify the user, and a marker is added to describe the nature of the events that have caused the transition (See Figure 3.2).

This transition activates a locking mechanism released by a timer. While the lock is held on the area, events dispatched for areas of the DOM unrelated to the stream will be aborted. Inactivity will expire the lock and switch back to unprotected mode. On the other hand, continued activity in a private area of the stream will renew the timer. In the future, we would reserve a CTRL-* key sequence to prolong the lock indefinitely and unlock it on-demand.

Activating the icon brings up information about the one specific locked area and its associated stream. Recall that there may be multiple private areas shown on the page for a given stream. The information shown for the stream would apply to all of the areas.

The monitor can simultaneously adorn four markers. A mouse click or keyboard event in a private area of the page will lock the monitor (with "M" or "K"

Figure 3.2: The Privacy Indicator "unprotected" (left), "protected" due to keyboard events ("K") in a private area (middle), and showing a security warning (right).

markers, respectively). In addition to text areas, Beeswax provides built-in support for private images. A mouse click in a private image area allows users to input new images. When the click is intercepted, Beeswax opens a file chooser dialog locked to the area. To differentiate the file chooser opened by the platform from one opened by the application, we add an "F" marker on the indicator. Lastly, a "*" marker attracts the user's attention to answer a prompt, e.g., to confirm sending or accepting an invitation or friendship.

3.3.4 Cryptography and Key Management

All keys are stored in local storage in the background page. Because Chrome's model is limited to a single store per extension, we avoid collisions of keys from different applications (but equal streamids) with a canonical naming scheme. To save/load a key to/from storage, we combine the key id (e.g., a streamid passed in the API) with the domain origin of the application, as well as the extension user's account name. The browser reports the tab URL associated with a content script's message port to the background page. This metadata and the naming scheme allow authorizing API requests access to only the keys associated with their application.

KEY AGREEMENT. As mentioned above, our platform supports an abstraction called a friendship channel. A friendship channel is a virtual private channel between the Beeswax extensions of two users. Beeswax has an API for one party to launch an authenticated key agreement protocol with another party. Assuming the parties have the appropriate public keys and Twitter userid bindings, and that the protocol completes, the two parties will possess two symmetric keys, one for encryption and one for authentication. Those keys will be used subsequently to encrypt-then-mac all communication over the friendship channel. Namely, two

friends use the channel to send each other stream invitations. Note that channels are between two users only, but streams may involve more than two users (see Section 3.2.1).

For the authenticated key agreement, we implement the well-known `AKEP1` protocol of Bellare and Rogaway [9] in the public key model. The JSON-like wire format of our protocol is in Table 3.2. We leave implementing a Diffie-Hellman style perfect forward secrecy key agreement protocol to future work.

KEY DISTRIBUTION. To publish a new set of keys, the extension posts three tweets. The body of the first contains `#encryptkey`, a time stamp, and the user's public encryption key. The body of the second contains `#signkey`, the same time stamp, and the user's public signature key. The body of the third contains `#keysig`, the same time stamp, the expiration date of the keys, a signature over the user's Twitter username and ID, the two public keys formerly identified, the same time stamp, and an expiration date (of about one week). The extension requires that the user be signed in to Twitter, so the extension can publish on the user's behalf.

The extension polls the user's Twitter feed to find tweets with the above markers and timestamps. For a given triple, the extension checks four conditions: the public keys in the body of the tweets are the same as the user's locally bound public keys, the time of the tweets is within `delta` (a configurable parameter in the order of minutes) of the time stamp in the body of the tweets, the time of the tweets is before the expiration time in the body of the `#keysig` tweet, and the signature in the `#keysig` tweet is valid. If any of these checks fail the extension raises an alarm to the user. If all pass, the extension binds the two public keys to the given Twitter ID, regardless of the state of an existing key binding.

KEY REVOCATION. A Beeswax extension subscribes to key announcements of other Twitter accounts. Revocation is achieved in Beeswax by publishing a new key over an old one, which causes other users' extensions to notice the change. Cases where certificates are absent, have expired, or have changed since the last validation are handled similarly. Whenever a certificate is determined invalid, associated keys (including stream keys obtained over invalidated channels) are kept, but marked invalid. The case of a certificate having changed differs only slightly in that there is opportunity to re-establish a valid friendship channel.

In our current model, the participants in a stream must trust the stream owner's extension to verify the validity of other participant's certificates, and re-key appropriately (e.g., new friendship requests and new invites). From the moment a stream participant's extension detects the stream owner's certificate being invalid, the stream cannot be used to encrypt new content. The extension provides the application with an API error code in this event. It is the application's responsibility to launch a new stream. A page is allowed to load with content from invalid streams (e.g., content encrypted prior to the invalidation), but the privacy indicator will display a lasting warning graphic (right-hand side of Figure 3.2).

CRYPTO LIBRARY. For cryptographic routines, we use the Stanford JavaScript Crypto Library [120] (sjcl) with AES and ECC support. The version of the library we use relies on the browser platform's `crypto.getRandomValues` to seed its pseudo random number generator (which is described in [60]). We have adopted sjcl because of its more mature ECC support vs WebCrypto [134] at the time the project started, but we could easily move to an alternate, native and faster, cryptographic library supported directly in the browser when it officially supports the primitives we need (ElGamal). Keys are stored in JSON format, using base64 encoding when appropriate.

We use elliptic curve cryptography (ECC) for all asymmetric material, in particular ECDSA for signing messages and ElGamal for encrypting small messages. Our symmetric encryption keys are 256bit long and we use the AES cipher in CCM mode (Counter with CBC-MAC, with 128bit IV and 64bit tag size). For HMAC, we use SHA-256 hashing, also with 256bit keys. All elliptic curve operations take place on NIST's recommended P-192 curve, which is available in sjcl. Moving to a potentially more secure curve, such as Curve25519 [12], or implementing ciphersuite negotiation is future work.

3.4 Evaluation

Beeswax aims to provide mechanisms and APIs that are fit for the development of modern web applications. To test this claim, we have used the platform primitives presented earlier on two applications. First, we transformed an existing web communication application, KiwiIRC (v0.9.0), into one that also offers encrypted

communications between groups of users. Second, we developed a new photo gallery application, PicSure, from scratch to demonstrate the abilities of the platform to handle richer media types. Below we discuss our experience in developing these applications and our platform's performance overheads.

3.4.1 Functionality and Experience

KiwiIRC is a web IRC client software package that contains both Web2.0 single-page client-side code, and server code to proxy messages between the browser and the IRC network. The IRC client already routed different types of messages between users and channels. It was therefore relatively easy to add another class of messages to carry Beeswax payloads within the IRC network. We changed KiwiIRC to route platform messages using information present in their headers. One difficulty we surmounted was that encrypted messages had to be fragmented to respect the length limitations of the underlying relay protocol and pass integrity checks.

The client was augmented with new commands:

`/joinenc [streamid]`: Takes a stream id (or creates one if unspecified), deterministically forms an IRC channel name from it, joins that channel, and allows typing new messages using the stream key.

`/inviteenc streamid [user]+`: Invites each user in the list to partake in the stream identified by `streamid`. Once a target user accepts the invitation, he/she automatically joins the conversation using `/joinenc`.

We have added graphical menu options to befriend other users in a chat room, as well as buttons to toggle encryption of input messages in the chat room panel.

When typing in regular messages, the application uses the arrow keys to recall elements from a history buffer and uses the "enter" key to signal the readiness for submission. The application cannot read keycodes from private areas, thus additional buttons are added outside the private area to recover this functionality.

For a seamless look, private areas hosting encrypted messages are styled like plain messages. That is, they are added with the other messages, along with information coming from the network (nickname of author, timestamp, etc.). The platform can apply highlighting to designated words in incoming messages, such

as nicknames, but we have yet to explore fancier formatting, as there is often a tradeoff between functionality and side-channel leakage.

The server-side code did not require any modification. The approximate number of lines of code added (not counting whitespace and comments) was around 400, constituting a 7% increase in code size. This gives an estimate of how little work is needed to add key-agreement and end-to-end encryption to an existing application using Beeswax.

RICHER SHARING. We have also built a web photo gallery called PicSure to show the capabilities of the platform to handle richer media types and allow application-defined sharing rules between users. It consists of approximately 2000 lines of script code, server and client combined. Put simply, it allows users to create albums of photos with simple descriptions and share individual albums with other users. All the data inside an album is part of the same Beeswax stream. The action of sharing albums is done through forms (drop-downs and lists) in the application. Unlike with overlay approaches, the user does not need to leave the application to define keys or assign them to the various objects. Rather, the application gives sensible sharing controls to the user: the owner of an album may invite any user currently online to collaborate on an album.

Editing descriptions is done with toggles and confirmation icons, like many edit-in-place applications, and uploading images is done using the Beeswax image file chooser. There are a few usability problems with the current platform, but they have solutions. First, having no capability to reduce images to thumbnails, PicSure renders album photos as full resolution images. Second, the sizes of the images are not known by the application, so it imposes a fixed size area to render them in the markup. A new appearance modifier could generate a thumbnail or resize the host element to match the image's aspect ratio. One other difference from a normal photo gallery is that the encrypted image data is retrieved and fed using data-URIs, as our implementation does not yet pass `src="{url}"` files through the decryption. This translates into encoding/decoding delays. Despite these usability issues, the application remains quite practical to share images quickly and securely.

3.4.2 Performance

We first report on the performance overhead of Beeswax, and conclude qualitatively with our own experience. The performance overhead of the platform as a whole can be attributed to several aspects:

Runtime initialization. Page load delays incurred by the initialization of the Page Runtime.

Events. Interception of every DOM event to prevent a possible data exfiltration (e.g., keycodes).

Message passing. Costs associated with passing messages between the page, content script, and background page.

Encryption. Cost associated with the encryption and decryption of user data.

The cost of initializing the runtime and intercepting the events is always present, even when the page in the tab does not use Beeswax. This is the base cost of loading the content script, injecting the runtime in the page, initializing the event hooks, and trapping on the first few load events. We compare load times of a bare bones HTML page with and without the extension enabled. This constitutes a base cost for loading any website. For $N = 400$, the minimal page's mean load time is under 65.5ms (σ=10.1 median=65 min=42 max=100) with the extension, versus 13.7ms (σ=4.7 median=14 min=5 max=25) without (cost average of 51.8ms).

Invocations of API methods do have an associated cost when generating events. This is because the Page Runtime needs to perform sanitization of these events. To calculate the overhead in event dispatch, we compare the time taken to generate and dispatch 1000 keyboard events. With the extension loaded, over N=100 rounds, 1000 events take on average 130.2ms to dispatch (σ=5.2 median=128 min=125 max=153) versus 51.8ms without (σ=5.2 median=51 min=46 max=64). Event dispatch takes 252% the time of the baseline, which is a considerable *relative* overhead, but in practice this is not noticeable.

The cost of message passing and encryption is predictable and is linear with the size of the plaintext submitted. Figure 3.3 shows where time is spent, by component, during calls to encrypt, i.e., with `get_cipher`.

The time spent in the Page Runtime includes the serialization of a message

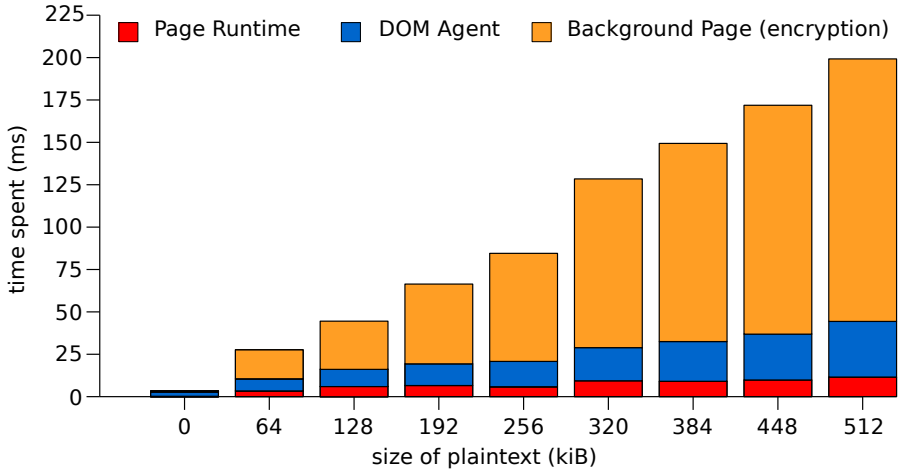

Figure 3.3: Median time (N=51) spent in each component when encrypting DOM elements, by plaintext size. Blocks (from bottom: Page Runtime, DOM Agent, and Background) do not overlap.

containing a node identifier (the one to encrypt) and 2 event dispatches with JSON serializations (on the call and on the return). The time spent in the DOM Agent includes reading the private DOM element's contents and passing the plaintext to the Background Page (and ciphertext back) over a message port (these ports support "structured clone" copies to avoid JSON serialization). Lastly, the majority of the time is spent in the Background Page performing the AES encryption routine and forming the stringified ciphertext (returned ciphertext objects contain a few parameters, namely an IV, which needs to be kept along with the text). Figure 3.3 shows median information over 51 runs. The top of a bar estimates the roundtrip time for encrypting private content of that size.

Overall, we did not perceive any performance impact in using the IRC client with and without privacy enabled. Similarly, we have run the extension on our desktops over the course of months without perceiving negative effects on non-Beeswax sites. A full user study is outside the scope of this paper.

3.5 Limitations

The API maintains data confidentiality, but this loss of visibility comes at a cost in interactivity for the application. Beeswax is not suitable for all applications. For instance, applications heavily relying on data-mining might find this cost prohibitive. However, from our experience building applications with the platform, we believe that many of the typical social-networking features such as comments, change notifications, "likes", replies, view counts, and untargeted ads (i.e., without access to content) can still be implemented. Also, where greater control is required over one data element (rich text format, image manipulation, etc.), appearance modifiers can fill in some of the gap. Lastly, applications which combine data with different confidentiality levels (public and private) might be drawn to Beeswax. Below, we detail more subtle restrictions and suggest improvements.

PRIVATE AREA INTERACTIONS. Our event sanitization conceals most interactions with private areas, but can be modified to let out more events. For instance, to allow the application to detect when an input image changes, Beeswax currently synthesizes a "change" event on the host element after the platform's file chooser is used. An improvement would be to interpret "Enter" strokes in private input boxes as an intent to submit, and synthesize a corresponding event interceptable by the application. Although arbitrary input validation is impossible without access to content, simple validators such as length enforcement could be communicated declaratively via DOM attributes. Leaving these decisions up to the application is an aspect where the platform approach shines over overlays.

The ability to drag-and-drop files onto private areas would be useful, especially for applications using photos, but difficult to implement securely. The intention of moving a file into a private area would need to be known ahead of time so that drag events could be suppressed along the way (to hide the file object from the non-private areas) until the file reached its destination.

RUNNING OTHER EXTENSIONS. The load order of multiple concurrently installed Chrome extensions is, to our knowledge, unspecified. A foreign extension could modify the browser's JavaScript runtime in unsafe ways (e.g., by exposing the original implementations of some global functions) and break our assumptions.

Whitelisting safe extensions to run concurrently with Beeswax is possible, but requires careful inspection. In Chrome's model, a permission exists to selectively enable or disable extensions. To ensure our extension loads first, a workaround is to disable all other extensions whenever a Beeswax-application is in use. If our API were part of the browser core, this limitation would vanish.

VISUAL FLEXIBILITY. Only CSS and "appearance modifiers" can be used to change the look and feel of private areas. We support basic word highlighting, which is admittedly limiting, but the platform approach shows potential. It could be extended to support markdown, BBCode, or other wiki-syntax display. There could be platform support to crop and rotate images, a platform-provided rich-text editor, etc. Like all web platforms and standards, we anticipate features would mature over time, guided by popular demand.

3.6 Related Work

We have looked at alternatives before settling on ShadowDOM to implement isolation in the UI. Namely, priv.ly [113] uses iframes, but their approach has two drawbacks. First, iframes are isolated by origin and behave like black boxes: it becomes difficult for the embedding application to manipulate and receive events for them. Second, the added dependence on another service or domain (in their case the content stores) to store and retrieve user data, as well as an ever-up-to-date look and feel (that has to match all websites), was unappealing.

The goals and limitations of overlays were discussed earlier. While they provide private UI channels, they do not defend against UI spoofing and require key definition and sharing to happen out-of-band from the application. Our architecture, with its integrated privacy monitor and in-band key distribution, allows a user to safely verify the sharing intentions of the application. In our case, this makes it possible to implement application sharing semantics which are in-line with those of the elements protected.

The data isolation mechanisms in ShadowCrypt [30] are similar, but not identical, to ours. We have found several design flaws in their latest implementation [119] which we detail below. Each of them allows an attacking website to exfiltrate the private data that ShadowCrypt is trying to hide. To their credit, their

code is open-source and receives suggestions for improvement. We note that since the project's inception, changes in Chrome's DOM core moved some object properties to getters/setters on the prototype chains [100]. This introduced breakage which has not yet been fixed, but the attacks listed below are valid regardless of the location of these properties (i.e., before or after said change).

ATTACK 1) IMPROPER ATTRIBUTE DELETION. Events targeting input boxes for encrypted content are canceled by ShadowCrypt. The decision to cancel an event is based on the target of the event, i.e., `evt.target`. During an event's dispatch, the target will change when crossing a shadow tree boundary. By introducing an additional shadow tree between the root of the document and the shadow tree for the "secure" input box, we can make an event appear as if it were targeting another element than the input box's host. The method `Element.createShadowRoot` is not deleted properly from the globals, allowing an application to mount this attack and let secure keypress events flow to the application. This attack can be mounted in two lines of code.

ATTACK 2) IMPROPER ATTRIBUTE DELETION. The property `shadowRoot` of elements gives access to Shadow DOM contents. ShadowCrypt deletes this property from elements at the moment they are added to the DOM, but this is too late. It is possible for a malicious application to create an element, redefine this property as being non-configurable, and add the element to the DOM after. The attempt to delete the property by ShadowCrypt will fail silently, leaving the "secure" contents readable by the application.

ATTACK 3) UNPROTECTED GLOBALS. So-called "deep" selectors can be used to retrieve elements across shadow tree boundaries. ShadowCrypt overrides the function `querySelector` to prevent selectors containing the substring `'/deep/'`, which would allow access to the hidden input element. However, it uses the global prototype method `window.RegExp.test` to perform the check. Because the method is accessed through the global `window`, a malicious application can redefine it to lie when a regular expression matches for `/deep/`, and later obtain a reference to the secure input element.

While attack 1) is simple, attacks 2) and 3) indicate that a rigorous construction

of the functions modifying the runtime are necessary. Properly implementing these defenses without bugs is hard. We believe the systematic way in which our Page Runtime is constructed helps with this, namely by avoiding the use of modified globals, and making assertions about the presence and absence of properties (e.g., a property should be there before deletion and absent after). Pushing our platform into the browser's core would make some of these defenses irrelevant, but for the moment, they are crucial.

There exists a body of work concerned with controlled JavaScript execution, which could allow Beeswax to make private areas less opaque to applications, but with possible privacy implications. TreeHouse [31] uses Web Workers to isolate scripts and provide them with a reduced view of the DOM, which is very much in line with our data protection strategy in Beeswax. AdSafe [77] and Caja [47] are language-based sandboxes that constrain language features and DOM access to allow for controlling the scope of effects of script execution.

Regarding hardening our Page Runtime against possible intrusions by the application code, there is also a wealth of research covering secure ways of isolating concurrently running JavaScript programs from one another. Many techniques offer ways of modifying JavaScript in safe ways for a containing page. Those include enforcing policies on scripts [45, 130] or at the language level [42] (for the older ECMAScript 3). They do fairly well in restricting the parts of the language used for safe operation, but the model of operation is reversed in our case. Our page runtime hosts an application whose programs cannot be modified (instead of an untrusted program that must conform to certain rules). As such these are not directly applicable to us.

The work on writing protected wrappers [42, 43], even though written for an older version of ECMAScript, still provides valuable formal descriptions of actual attacks. The event models (i.e., sanitization) in the browser are, however, not described in this previous work. Some the problems, e.g., protecting the globals against tampering, are addressed previously in work on JavaScript compilation [27], but for it to be useful would require rewriting our Page Runtime in ML.

Our work also relates to work on information flow control. In COWL [61], the authors make use of the existing isolation in browsers (contexts) to control flow between domain origins. A finer-grained labeling scheme than per-origin would be

required to apply to our needs.

3.7 Discussion and Future Work

Our simple invite API could be improved to support other kinds of groups, such as closed groups or groups where all existing parties need to agree before a member can invite a new person. Assuming a user trusts another friend (and their extension) to forward invites, then this could allow knowing the full extent of the users with whom the keys have been shared. Lastly, our streams can be improved to defeat replay attacks from the application. At the moment, a malicious application could reorder and resend a stream's messages. We plan to fold in sequence numbers as authenticated metadata and display them in the privacy indicator as a simple defense.

Ideally, the platform would support applications for both desktop and mobile. Popular mobile platforms may be trustworthy, but are not as open and extensible as desktop environments. There currently is not good support for third-party extensions on mobile web browsers. We do not eliminate the possibility of adapting our techniques to an OS kernel and/or rendering APIs, but we suspect it would require a substantial rewrite. By reusing an existing mobile browser engine customized with our platform tools, we may support web-based applications designed for mobile.

3.8 Conclusion

The Beeswax platform allows developing interactive, multi-user web-based applications. It balances the desire of developers to maintain control over the look and feel and functionality of their applications with the users' desire to know and control who has access to their private data. The Beeswax API forms secure communication channels between users and prevents data exfiltration by the applications. Beeswax allows the community to focus its scrutiny on just the platform, instead of all applications using it. Like all platforms, Beeswax's functionality is not set in stone. We anticipate that Beeswax will evolve to accommodate developer and user needs, and either move into the core of one or more browsers or be maintained by an open-source community.

Table 3.2: Alice contacts Bob for the first time and negotiates a shared secret with him using our KAP (via `get_friend`). Messages flow between users' extensions in the direction of the arrows. Alice's keys are denoted by the letter A, and Bob's with B. Key subscript $\cdot_s$ denotes signing, and $\cdot_e$ encryption. Public keys are suffixed with $+$, private ones with $-$. Functions `add_sign` and `add_HMAC` compute signature and HMAC (respectively) and return their augmented input.

Alice		Bob

Input: (A_s+, A_s-, A_e+, A_e-), (B_s+, B_e+)
• Generates A_{FID}: contribution to friendship ID (challenge nonce).
• Sends `KAP_MSG1` Message.
	→
	`type: KAP_MSG1,`
	`hdr: { To: "B", From: "A",`
	`AFID: $A_{FID}$, BFID: "" },`
	`payload: null`
		Input: (B_s+, B_s-, B_e+, B_e-)
		• Receives `KAP_MSG1`
		• Finds A_s+ and A_e+ if unknown (See Section 3.2.2).
		• Validates `To` .
		↓

• Receives `KAP_MSG2`.
• Validates `hdr`.
• Validates signature with B_s+.
• Decrypts `payload` with A_e-
• Saves plaintext as MK.
↓
	←
	`add_sign(k=$B_s-$, {`
	`type: KAP_MSG2,`
	`hdr: { To: "A", From: "B",`
	`AFID: $A_{FID}$, BFID: $B_{FID}$ },`
	`payload: enc(k=$A_e+$, m=$MK$)`
	`})`
		←
		• Generates B_{FID}: contribution to friendship ID (challenge nonce).
		• Generates MK: the master key of the friendship.
		• Replies with `KAP_MSG2` message.

• Sends `KAP_MSG3` message.
• Derives & stores friendship keys:
F_{mac}= `HMAC(`MK`, "mac")`
F_{enc}= `HMAC(`MK`, "enc")`
	→
	`add_sign(k=$A_s-$, {`
	`type: KAP_MSG3,`
	`hdr: { To: "B", From: "A",`
	`AFID: $A_{FID}$, BFID: $B_{FID}$ },`
	`payload: null`
	`})`
		→
		• Receives `KAP_MSG3`.
		• Validates `hdr`.
		• Verifies signature with A_s+.
		• Transitions to A-has-friendship-keys state
		↓

• Receives `KAP_MSG4`.
• Validates hdr.
• Validates HMAC.
• Transitions to B-has-friendship-keys state
	←
	`add_HMAC(k=$F_{mac}$, {`
	`type: KAP_MSG4,`
	`hdr: { To: "A", From: "B",`
	`AFID: $A_{FID}$, BFID: $B_{FID}$ },`
	`payload: null`
	`})`
		←
		• Derives & stores friendship keys:
		F_{mac}= `HMAC(`MK`, "mac")`
		F_{enc}= `HMAC(`MK`, "enc")`
		• Sends `KAP_MSG4` Message.

Chapter 4

DistoRt: Anonymous Communications on Existing Infrastructure

4.1 Motivation and Overview

Anonymous communication is a long sought goal of dissidents and privacy advocates. Promising anonymous communication models have been proposed, based on IP-multicast, mixes (e.g., Vuvuzela and Alpenhorn [37, 67]), DC-nets and custom broadcast networks (e.g., Riposte [17]), and Private Information Retrieval (PIR) databases (e.g., Pung [4]). Unfortunately, the vast majority of these systems have not gained wide adoption outside academic circles, and require new large-scale communication infrastructure to be built in order to achieve a non-trivial amount of anonymity. The lack of business rationale, combined with the pressure to instaurate mass-surveillance capabilities, might hinder the development of such infrastructure.

Fortunately, Tor [23] demonstrates that a practical anonymity system can be built on top of existing, economically viable, IP-routing infrastructure. Tor strives for full anonymity, even when not all of the routers are trusted. However, for an end user it is difficult to reason about the threat of de-anonymization when using Tor.

For instance, by penetrating enough routers [32], or by controlling both entry and exit nodes [62], communications can be de-anonymized on a given path [35, 69]. The security of the network is a legitimate concern: suggestions of attacks on the Tor network has surfaced in the past [109]. This concern is amplified since an end user likely has little visibility into whether the routers her Tor client selects for her various circuits are currently penetrated by an untrusted party.

Our wish is for users to be able to form *ad hoc* anonymity groups in which they can send each other confidential messages. To the best of our knowledge, there is currently no deployed system which satisfies this goal. Tor, and previous anonymity networks, provide some of the technology to attain our goal, but they do not provide us with a complete solution. For instance, they do not provide a clear path to key distribution and user authentication, which are crucial to achieve practical user-to-user communication.

RECEIVER ANONYMITY VIA MULTICAST ENCRYPTION. Inspired by Tor, the main goal of this work is a system for anonymous communication that, like Tor, is built on existing infrastructure and widely deployable. However, we aim for a threat model that is easier to reason about than Tor's for the typical user. The essential mechanism of our system DistoRt is not novel: it is based on multicast encryption. When user A sends a message to user B, she encrypts her message, along with an integrity check, with the public key of user B, and then broadcasts the cipher text over a multicast channel. All the users on the channel decrypt the message and perform the integrity check. User B will properly retrieve the message while other users will rightfully discard the message with high probability. We use a standard hybrid encryption scheme known as ECIES-KEM [59]. The public key encryption component of this scheme is ElGamal over an elliptic curve. ElGamal, as well as the hybrid scheme have an important property for this application, key-privacy: the ciphertext does not reveal the public key used to create it, up to the intractability of the Computational Diffie–Hellman problem [11, 58].

Clearly, an adversary with access to all of the traffic associated with such an encrypted multicast can ascertain the identity of the sender, from where and when the multicast was sent, and the identities of the entire set of recipients of the multicast. The listeners on the multicast channel constitute the anonymity set for the

channel. In spite of the adversary being able to identify the sender, the message traffic does not provide additional knowledge about the intended recipient among the anonymity set, beyond the adversary's *a priori* knowledge, up to the above computational assumption. That is, the multicast encryption mechanism described is designed to achieve *receiver anonymity*.

We assert that receiver anonymity is of great value in many situations. A sender may well be comfortable with the network adversary knowing that he sent a message as long as the adversary does not know the contents or the recipient from among a large set. Note that the network adversary similarly knows that a Tor user has sent an encrypted IP packet.

LEVERAGING AN EXISTING, SCALED SERVICE. Hordes [40] anticipated the adoption of IP-level multicast routers as a means to achieve receiver anonymity. As IP-level multicast is not widely deployed, and is not likely to be widely deployed in the near future, Hordes is not viable for most users. To achieve immediate deployability and scalability, we have built DistoRt as an overlay on Twitter. Any other service that supports a sufficiently rich interface for supporting multicast or publish/subscribe would also work.

THREAT MODEL 1.0: MAN-IN-THE-MIDDLE ATTACKS. DistoRt assumes that Twitter can act maliciously and insert a message as if it were posted by any Twitter user. We discuss our mitigations of Twitter's power as a man-in-the-middle attacker in Section 4.3.4.

We assume that a party (other than Twitter) that is not in possession of given user's credentials (password or OAuth credential) cannot post a tweet under that user's Twitter handle. More generally, such a party cannot mount a MITM attack on the connection between Twitter and Twitter users.

TIMING ATTACK MITIGATION WITH COVER TRAFFIC. Inevitably, a user will need to reply to a message received anonymously with a multicast encrypted message of her own. However, an immediate reply may leak that she was the intended recipient of the original message. To avoid timing attacks, every user client in DistoRt sends exactly one multicast encryption every set interval. If there is no actual message to send in that interval, the client sends a bogus multicast encrypted

message as cover traffic.

Mitigating timing attacks in this standard way is expensive for large anonymity groups. Our goal is to support anonymity groups of size, e.g., 10,000. Given the size of tweets, this would translate into network usage of 1.25GB/day for listening to DistoRt traffic. For some users, this may consume a large chunk of their monthly data service plans' bandwidth caps. To allow users with bandwidth restrictions to participate in large groups DistoRt provides a mechanism that tradeoffs the amount of listening bandwidth consumed with the time required to receive a message. This is described in detail in Section 4.3.3.

PUBLIC KEY DISTRIBUTION. A second major component of DistoRt is in-band key management. In order for A to send an encrypted multicast to B, A must know B's public key. Public key distribution is particularly challenging in this setting for two reasons. First, user A searching or querying for a certificate of another user potentially leaks information about with whom A wishes to communicate. We believe that any system for anonymous communication that does not include key distribution is simply sweeping a crucial issue under the rug.

Second, the threat of a spoofed certificate in multicast encryption is more severe than in point-to-point communication. In the latter, a spoofed certificate for an encryption key in a system that still properly routes based on the identities in the certificates results in a user receiving a message that it cannot decrypt. But for multicast encryption, the encryption is essentially doing the routing. Thus a spoofed certificate effectively routes encrypted messages to the adversary who then decrypts them.

Existing approaches to public key management are problematic for this application from a number of perspectives. One such approach is the PGP public key database. However, as no authentication is required to insert a certificate into the database, an adversary may insert a public key for a targeted identity. For multicast encryption this is particularly dangerous. While a human may be able to inspect all of the keys for a given identity in the PGP database and winnow the legitimate keys from the chaff, properly identifying legitimate keys programmatically in an adversarial setting was out of scope for this work.

Another approach to public key management would be to require users to get

certificates for their keys from a certificate authority. However, we believe that asking users to get certificates from a CA creates a barrier that severely limits adoption. Moreover, there is still the problem of distributing the certificates. Putting them in a database and having users query the database leaks information. The certificate database could be a PIR database to achieve anonymous retrieval of certificates [4, 41, 49, 63]. However, we assert that reasoning about the trustworthiness of a given CA, let alone a sequence of CAs in a certificate chain is at best complex for a typical user. This is true even if those certificates are backed by Certificate Transparency (CT) logs (For example, a rogue CA can still push certificates into CT logs. This will be caught by consistency monitors but the path to notifying the end user is not clear cut.)

Another approach to key distribution is provided by Beeswax [39], which uses Twitter for key distribution. A user client generates, or imports, a key pair, creates a self-signed certificate with the user's Twitter handle, and posts the cert periodically as a tweet. Certificates are fetched when connections are established. Clients reject certificates if the handle in the certificate does not match the handle of the tweet or the certificate does not otherwise validate.

A malicious third party cannot spoof clients into accepting a bogus public key for a target user handle since the malicious party cannot inject a post with the target user's handle.

Beeswax assumes that if Twitter injects a spoofed tweet supposedly authored by a given account, all followers of, or queries for, that account would read the same spoofed tweet. Such spoofing by Twitter is easy to detect. Beeswax clients simply listen for their own certificates. If they read a certificate with their own handle that they themselves did not post, they balk.

THREAT MODEL 2.0: EQUIVOCATION. We take an approach similar to Beeswax but with two important modifications. First, Beeswax key distribution is tightly linked to client relationships and so leaks such relationships. In contrast, DistoRt clients passively listen on a Twitter stream to which all clients periodically post their certificates.

Second, Beeswax assumes that Twitter does not equivocate in its attempts to spoof users. However, given the consequences of a spoofed certificate for multi-

cast encryption discussed above, it is prudent to make as few assumptions about Twitter's potential malicious behavior as possible. DistoRt assumes that Twitter may *equivocate*. That is, Twitter is not bound by consistency between feeds and query responses. Twitter can craft a sequence of potentially spoofed tweets specific to the account consuming the sequence. With this assumption, Twitter can evade the mitigation in Beeswax by sending a spoofed certificate for a given user to all relevant accounts *except* the account of the user being spoofed.

To mitigate against equivocation, DistoRt employs a second service that acts as an independent certificate service. A client will not install a user handle + public keypair 2-tuple into its validated database unless the self-signed certificates retrieved over both services agree. For the instantiation of DistoRt presented here, we build the independent certificate service on GitHub. We assume at most one of GitHub or Twitter to equivocate on the certificates of a given user. We describe the full key validation protocol in Section 4.3.1.

CONTRIBUTIONS. Overall, DistoRt is designed from the start to be readily deployable and widely adopted. To that end, it is built on top of existing, scaled services; it has in-band key management; and it only requires users to have accounts on Twitter and GitHub and to download and install the DistoRt extension. In addition, DistoRt is designed for a threat model that is easier to reason about than Tor's. Finally, DistoRt simultaneously allows for large, user-launched anonymity groups, cover traffic within each group, and the ability for each user to set a tradeoff between bandwidth consumed by their DistoRt client and time to receive messages within an anonymity group.

4.2 Components, Assumptions, Goals

4.2.1 Components

We summarize each of the components in our design, so as to provide a whole-system view of DistoRt, before we delve into its inner workings.

PUB/SUB SERVICE: TWITTER. We use Twitter as the single network communication router for our messages. Messages sent by DistoRt are formatted as text-only

140-character tweets. Each DistoRt user runs a client which connects to Twitter using Twitter account credentials only – there is no separate account for DistoRt. Twitter handles thus form the basis of our addressing scheme. We expect users of DistoRt to configure their clients with reputable accounts, as the handle (and underlying IDs) are rooted in the trust chain, but this is not a technical requirement.

The DistoRt client will use Twitter for sending two types of payloads: self-signed certificates, and free-form messages addressed to a single recipient, which we call *twists* (to disambiguate from tweets). The self-signed messages are used both to announce identity (with keys), and group memberships.

SECONDARY CERTIFICATE SERVICE: GITHUB. To protect users from possible equivocation by Twitter on their certificates, we rely on a secondary service. Briefly, certificates received on Twitter must be matched with a copy on GitHub before being accepted by clients. The DistoRt client is therefore also configured with a GitHub account handle. When the client posts a new certificate (e.g., key rollover), a repository on GitHub on that account is updated with the new information. Similarly, after a certificate from another user is discovered on Twitter, it will eventually be verified on GitHub. The name of the secondary account is deterministically derived from the first.

CLIENT-SIDE DISTORT APPLICATION. Our application provides end-to-end crypto without adding new server-side infrastructure for performing key distribution or messaging. It can be installed as a browser extension for Google Chrome.

TWITTER STREAMS AND DISTORT ANONYMITY GROUP NAMES. Users join anonymity groups. Messages sent by the client are broadcast to other members of the group. The notion of DistoRt groups is completely disjoint from Twitter followers, and participating in DistoRt does not require changes to the social graph of the user. To keep messages circulating within one group, we "route" twists with Twitter hashtags when posting, and filter on these tags when "streaming" twists. "Streams" are long-lived HTTPs connections on which Twitter delivers live tweets matching a rich query. The query syntax covers, namely, presence of one or more specific tags in tweets. Our tagging is such that the twists can be routed appropriately with the Twitter streaming API.

PUBLIC KEY PAIRS AND CERTIFICATES. The client generates (or imports) and stores two keypairs: one for signing and one for encryption. We use ECC because of its compact size and because of the receiver-anonymity properties of EC ElGamal. Public keys are included in an ECDSA self-signed certificate generated at the client. The certificates explicitly include the identity of the user: Twitter handle, Twitter ID, and GitHub handle. They also advertise the group names in which the user takes part, as well as their subgroup. The user selects the subgroup according to how much bandwidth they are willing to consume listening to the group stream. This is covered in Section 4.3.3.

4.2.2 Security and Design Goals

The following are the goals for our design of DistoRt.

MINIMAL IMPEDIMENTS TO DEPLOYMENT AND ADOPTION. Our design has been primarily driven by the goal of developing a system that has the potential for widespread adoption and for scaling to a large number of users.

THREAT MODEL COMPREHENSIBILITY. Given the difficulty for a typical user to reason about the security of their Tor circuits, a primary goal of this work is an anonymity system where messages pass through only a small number of components and the security assumption of the components are clear.

SCALABILITY TO LARGE ANONYMITY GROUPS. We must support anonymity groups of size between one and tens of thousands, which a user can join and leave freely.

TIMING ATTACK MITIGATION. The DistoRt client must generate regular, periodic cover traffic. Twists with actual payloads (e.g., originating messages or replies) are queued up and slotted into the cover traffic schedule.

SUPPORT FOR CLIENT-SIDE BANDWIDTH LIMITATIONS. Cover traffic for large groups can be significant. DistoRt must provide a mechanism for users to adapt to the client-side bandwidth requirements of participating in large anonymity sets, even if it comes at the cost of lower throughput for those clients.

MINIMAL OUT-OF-BAND REQUIREMENTS. A chief goal of DistoRt is to minimize out-of-band requirements for users. DistoRt is designed such that users only need to establish trust in the binding between a Twitter handle and an actual person. (GitHub handles are derived from Twitter handles. See Section 4.3.1) Each user is responsible for establishing his own desired level of trust via out-of-band means. Users should be aware that the online means taken to establish that trust may leak hints on intended communication partners to a network adversary.

To improve adoption and to reduce the threat of traffic analysis via *ad hoc* key discovery, DistoRt is designed to provide in-band key distribution. Certificates are designed to bind Twitter and GitHub handles to public keys.

CONFIDENTIALITY, AUTHENTICITY, AND ANONYMITY. The focus of this work is building a functional, easy to use system with the potential for widespread adoption. DistoRt does not make any claims to cryptographic novelty. We use standard cryptographic mechanisms in standard ways and DistoRt inherits the security properties of the underlying mechanisms. As discussed in Section 4.3.4, the confidentiality of twists is derived from the security of ECC encryption with curve P192. The security of the signatures in our certificates and our twists is derived from the security of ECDSA signatures with the above curve. Receiver anonymity is provided by a property of ElGamal encryption where a cipher text does not reveal the public key used to produce it [11].

4.2.3 Threat Model and Assumptions

SECURE SERVER AUTHENTICATION AND TLS CONNECTION. We assume that the Twitter domain is properly authenticated in the user's browser and that the TLS connection is terminated by a Twitter server. As noted, DistoRt is built as a Chrome extension. Chrome comes with the Twitter certificate hardcoded via HTTP Public Key Pinning (HPKP).

For GitHub, a fingerprint of the expected certificates can be encoded into DistoRt and checked against the HPKP response headers for a match.

After TLS connection establishment, we assume that a user's browser is connected to the Twitter or GitHub domain over a secure HTTP connection that pro-

vides confidentiality, authenticity and replay protection (e.g., the channel is immune to a MITM attack).

SECURE USER AUTHENTICATION AND AUTHORIZATION. Excluding Twitter, we assume that only a party in possession of a user's credentials (password or OAuth token) can post or read from that user's Twitter account. We treat everyone who has the password for a service account to speak authoritatively for that account, as if the account was controlled by that person. A malicious party may obtain a Twitter (or GitHub) user's password, and in such a case we make no claims about the security of DistoRt. We do, however, discuss recovery from password compromise in Section 4.6.

SPOOFING AND EQUIVOCATION BUT NO COLLUSION. Twitter, of course, may make reads and writes from and to any user account. Twitter may, for example, use this ability in an attempt to spoof a twist or to spoof a certificate. Moreover, we assume that in an effort to subvert the security of DistoRt, Twitter may *equivocate*. For example, Twitter may present a different tweet, albeit one with the same user handle and time stamp, to different followers of the same user, or to different subscribers of the same stream. As noted earlier, it is unlikely that Twitter currently has the ability to equivocate and it would require a significant engineering effort to develop such an ability. Nonetheless, given the negative consequences of a successful spoof, it seems prudent to place as little trust in Twitter as practical.

We make the same assumptions for GitHub. However, we assume that at most one of the two services is equivocating about the posts of a given user. In the case that one service is equivocating, the other may still attempt a spoofing attack without equivocating.

Finally, we assume that the two services do not collude to thwart the security of DistoRt. Twitter and GitHub are sufficiently independent commercial organizations that the organizational and engineering barriers against collusion are extremely high.

TRAFFIC ANALYSIS. We assume a network adversary capable of amassing all of Twitter's traffic. We assume the same adversary may track other communications, such as email, of the users of DistoRt. Of the possible traffic analysis threats, we

believe that ascertaining the public key certificates that a given user is collecting is extremely valuable information. Hence, attempts to do so are within the scope of the network attacker assumed for this work.

Twitter itself, of course, can track all data and metadata uploaded to and downloaded from its service, and all API calls made. For instance, Twitter has full knowledge of tweets retrieved and stream subscriptions. A similar statement holds for GitHub. For the purposes of traffic analysis we assume that entities may share traffic data.

CLIENT-SIDE TRUSTED COMPUTING BASE. We assume that the operating system and web browser of the user are part of the trusted computing base for DistoRt. We further assume that the DistoRt client-side web extension properly isolates private keys from other processes and network interfaces. That is, access to private signing or decryption keys is only possible through physical possession of a device. We offer no protection to a user between the time a key is compromised and the compromise is discovered. We outline in Section 4.6 how to reset certificates after a compromise is discovered.

Currently, we assume that all users run our client side code and do not attempt to push more than their fair share of messages through the system. This is not unreasonable, because any client (misbehaving or not) is ultimately bound by Twitter's enforcement of rate-limits. As future work, individual clients could monitor the send rate of the other users in its anonymity group and delete messages from misbehaving users as a disincentive.

4.3 Architecture and Implementation

DistoRt users require little information from each other to communicate: trust starts with the identity of a user on the primary service, as well as the knowledge that this user runs a client participating in the protocol.

4.3.1 Key Distribution

A user's primary identity is the Twitter handle (e.g. @bob). Twitter handles can technically be changed by the user, although is ill-advised when the handle itself

is the only reputable recognizable identifier, and basis of trust for human users. By including both handle and ID in certificates, it is possible to revoke active certs if an account changes handle, or if it is assumed by a different account.

First-Time Users of DistoRt

The primary and secondary accounts are to be linked by the client. The GitHub account will store a second copy of the client's cert. We trust that, for both the primary and secondary services, only entities in possession of the account credentials are able to post new data to their feed.

In our certificate verification protocol (Section 4.3.1), a certificate found on the primary is expected to match a corresponding post on a secondary account. If more than one secondary account were to present a certificate for that user, i.e., a self-signed certificate claiming to come from @alice was found in two secondary accounts, then the client should be in a position to unequivocally pick the account owned by @alice.

Of main concern here is detecting a possible equivocation attack launched by the primary service. That is, we must guard against the possibility that the self-signed certificates @bob has retrieved from @alice's feed may differ from those genuinely posted by @alice. In this particular case, @bob would likely find two different self-signed certificates on GitHub, both claiming to be for the primary handle @alice.

We choose a simple strategy for linking the two accounts – to reserve a name on the secondary uniquely derived from the primary account name (in a way that prevents at least accidental existing-name collisions), e.g. linking @alice on Twitter to DistoRt-alice on GitHub. One problem with this approach is that Twitter users joining DistoRt might find their desired GitHub name already taken. We could have allowed arbitrary names on the two accounts instead. But, we felt that the convenience of addressing users with a single handle rather than a 2-tuple (e.g primary + secondary) outweighed the (unlikely) pain of discovering one's derived secondary name unavailable. There is not otherwise a significant technical difference between the two approaches.

Thus, we note that in some cases, joining DistoRt will require either renaming

the handle of the primary account (which luckily, in the case of Twitter, preserves all followers), or creating a new pair of accounts. GitHub's user policy specifically forbids name-squatting [95], which indicates that users believing to be victims of name-squatting could instead recourse in contacting the service's support.

Posting Keys

Clients publish their public keys to both the primary and secondary in the form of self-signed certificates. The two keys are an encryption key and a signing key, both posted in the same operation, and signed together with the signing key.

The information included in the signature is listed in Table 4.1. Fields of the certificate may not be included in the certificate themselves, but as part of the surrounding metadata, to save space. For instance, we omit the Twitter handle and account ID from tweets, because Twitter includes them as part of the metadata in every tweet. Information omitted is however included into the signature.

Our self-signed certificates take up around 200B of space, a small figure comparison to the X509 SSL certificates used on the web. For instance, simple X509 domain-validated (DV) certificates issued by the CA LetsEncrypt, even in their compact binary DER form, occupy more than 1600B – excluding any intermediate certificates which would have to accompany the leaf cert. By using base16k to encode binary data, we can fit certificates on Twitter in a single 140 unicode (NFC) character tweet.

Clients will periodically perform "self-health" checks to ensure their own certificate's validity online. The certificates produced have an expiry set to about one week, and clients are instructed to refresh them every 3 days. Active clients will renew usage of their key by re-posting the same key with a new expiry date. If more than one key is posted within a period, then clients will only consider the latest set of keys valid.

The steps taken to post keys are as follows:

1. Record current time (`validFrom`)
2. Set expiry as one week past (`validUntil`)
3. Assemble certificate fields, as described in Table 4.1

Table 4.1: Information stored in self-signed certificates.

Name	Description	Size (B)
`groups`	Group Memberships	42
`pri_id`	Primary service account ID	8
`pri_hdl`	Primary service handle	15
`sec_hdl`	Secondary service handle	39
`validFrom`	Start time (unix UTC)	6
`validUntil`	Expiry time (delta secs)	3
`enc_pub`	ECC Public encryption key	24
`sign_pub`	ECC Public signing key	24
`signature`	ECDSA Signature	48
	Total	209

4. Post to GitHub, in repository `{sec_hdl}/distort-app`. This step implicitly invalidates the previous cert posted – only the latest cert posted is usable. The repository is automatically created if it does not exist.
5. Post to primary service, Twitter, with certificate hashtag `#dtcrt`. Other clients will learn about the new certificate at this time.

The self-health checks determine whether the certificate needs to be reposted. The expectation of the check is that both services agree on the latest cert, and that the keys in the cert are recognized by the client.

1. Retrieve for the user's certificate on the primary. This is a search query on a Twitter feed.
2. Retrieve the user's certificate on the secondary. A partial GET request to retrieve a file from a user's repository.
3. If any certificate is missing or invalid, repost.
4. If the certificate's validity is past its half-life, repost.
5. If the key in the certificate is not the key stored in the client, raise an alarm for the user. This will happen if the client is reinstalled, or an entity other than the client has modified the online content.

Learning Keys

The verification for incoming certificates received requires no manual interaction, but requires network access to external services (namely GitHub). To preserve anonymity, the verification process should not apply to only accounts of interest to the user – a network adversary could infer user interests if only a select handful of public keys were verified. This doesn't mean that *all* certificates need to be verified however – a daily limit could be set, and verifications could be queued for a later time.

DistoRt clients continuously stream for incoming key publications on the tag `#dtcrt`. Streams are Twitter's mechanism to select tweets of interest by hashtag, user, or keywords. The client extracts certificates from received tweets for verification. The result of a successful verification is to add the public key into a local cert database in the client.

The first step of the verification will validate the information as seen on Twitter. If any step of the verification fails, the certificate tweet is ignored:

1. The envelope should match the contents: `pri_id` and `pri_hdl` should match the author of the tweet.
2. The certificate time `validFrom` should be within 5 seconds of the tweet's advertised post time. The end of the validity period `validUntil` should be set to a date in the future, but no more than a period than one week plus one post interval, i.e., 10 days, to encourage short-lived keys.
3. If a previous key binding was accepted for that user, then `validFrom` should be larger than the previously known value for `validFrom` that was accepted. This forces `validFrom` to be monotonically increasing for a user.
4. Verify the signature of the self-signed cert. The contents of the cert still needs to be checked against the secondary service, as detailed below.

Passing the steps above yields a binding between a Twitter handle, and two public keys (encryption, signing). To detect equivocation attempts from the service provider on that user, we complete the verification by looking for a certificate match on GitHub. Again, if a step fails verification, the post is ignored.

1. Verify that `sec_hdl` is the authoritative mirror of the primary account, i.e., that the secondary service handle `sec_hdl` is the valid unique derivation for the primary handle `pri_hdl`. If indeed the user behind the primary handle `pri_hdl` was participating in DistoRt, then the `sec_hdl` has been reserved before keys were posted, assuming that GitHub usernames remain unique amongst all their users.
2. Issue a GET request to the secondary account (`sec_hdl`) to retrieve the latest copy of the certificate. Our prototype retrieves the certificate file from repository `{sec_hdl}/distort-app`.

   The repository must be owned by that same account. The rules on the service are such that write access to the repo is controlled by the owner of the account. Therefore, content on the repository is either explicitly uploaded by the owner, or by the service.
3. The certificates retrieved on each service are equivalent (same keys and fields), and the signature verification passes.

This completes the certificate verification process. Verified certificates are stored in the client database. It would be straightforward to implement a user "contact list" feature, and store only the certificates which are of interest.

By listening on the key channel a client can, in a post interval's time, learn keys from all active users. If 100K users participate and share their certificate every 3 days, this amounts to roughly $\approx .39$ certificate posts per second $(100000/(3600*24*3))$, i.e., one verification every 2.6 seconds on average. As of May 2017, Twitter processes over 7500 tweets per second [127], which is 20000x higher. A popular hashtag, #SuperBowl, is used 3 million times in a 5 hour period: 167 tweets per second or 432 times the certificate-post rate.

4.3.2 Anonymity Groups

The greater the number of participants, the greater the anonymity. However more participants necessarily means higher message volume, and more work for clients. Clients are all placed in the same default group upon installation, but users can order their clients to form, join, or leave groups. DistoRt groups are, conceptually, broadcast channels with names. They take the form of a Twitter hashtag. When

posting, messages are routed to a group by including that hashtag in the message. When receiving, clients filter on the tags with the streaming API.

We initially believed that it would be more practical to overlay DistoRt group broadcast on top of Twitter follower notifications, but it proved difficult to arbitrarily change follower graphs of the user from the application. Namely, Twitter imposes ill-defined limits on the number of followers and followees that users may have (and on their ratios). We have also considered creating groups using hierarchical cascading graphs of followers, or adding intermediary nodes to form "group hubs" (supernodes), but these approaches had different theoretical limitations (namely reaching maximum number of posts per day/period), and were logistically difficult to setup (e.g., creating new accounts for each new group). The simple approach of having Twitter as the central hub is easy to understand, and still allows each client to stay within the limits of the ToS.

To join an anonymity group, a client will subscribe to receive tweets made to that group – this implies opening a long-running streaming connection to Twitter on the corresponding hashtag. Then, the client will initiate the key-posting and re-posting process with the group added to the certificate, as described in Section 4.3.1. This process is repeated for each group the client has joined.

Clients participating in a group will receive key updates and messages from other group members. They can stop receiving group messages by leaving the group. To do so, it suffices to end the message stream on the group channel – we do not currently send a message to notify others about parting with the group.

To accommodate the varying network bandwidth constraints of participants in a group, we carve each group into two equal subgroups, recursively, forming a binary tree of subgroups (Figure 4.1). Clients not in the capacity to receive whole-group traffic can elect to tune on a fraction of it by subscribing to a subgroup commensurate of their network and computing "budget". By subscribing to a smaller fraction of the total group traffic however, clients trade off network bandwidth for increased latency for receiving messages posted by peers (Section 4.3.3).

The precise listening position of a client within an anonymity group tree is specified in the certificate anonymity group list. We reserve as many bits as the height of the tree in the last unicode character of the hashtag to encode a client's listening subgroup.

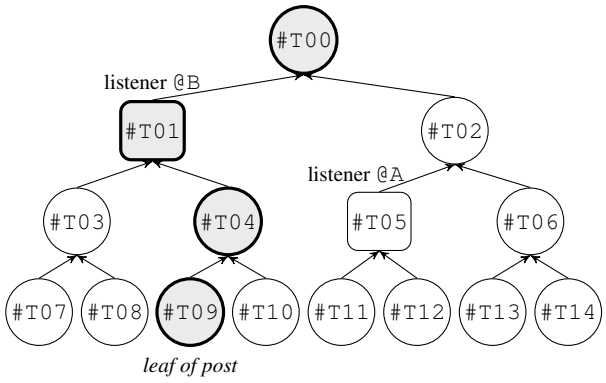

Figure 4.1: Organization of anonymity group `#T` into subgroups (4 levels), and participation of two members of the group. Client of user `@A` posts to node `#T09` – thus reaching out to all clients listening on any darkened node.

Clients initially join the root node of the group, which captures every message sent in the group. Clients continuously track the amount of incoming traffic at their current position, and can dynamically reposition themselves up or down to approximately double or halve the receiving rate, respectively, and then post a certificate about their new position. When downsizing, clients can coin-toss to descend left or right, to help achieve balance between sibling subgroups.

Performing network sensing for a short time lapse before moving between subgroups is possible, both to minimize iterative guesswork, and perhaps keep the group more balanced globally, but we leave this to future work. Similarly, it would be possible to occupy multiple non-overlapping subgroup positions to receive non-power-of-2 portions of the traffic, but we chose simplicity over finer control.

4.3.3 Sending Twists

The certificates indicate the subgroup, i.e., the listening disposition of a user within a group. For a message to reach a recipient, the message must be sent to a subgroup on which the receiving client is listening. Twitter does not implement "hierarchical" hashtags, to our knowledge. The streaming API has a rich syntax which would allow clients to filter incoming tweets based on unions of intersections of hashtags, but it is limiting in the number and length of search terms that can be specified.

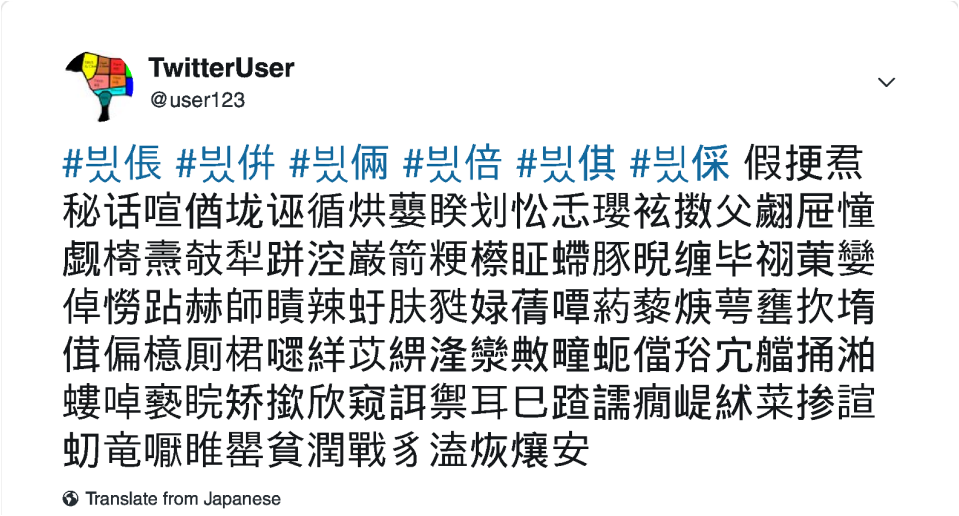

Figure 4.2: Screenshot of a twist on Twitter.

The approach we take to simulate hierarchical groups is to specify an enclosing set of subgroups in each twist. This takes the appearance of a series of hashtags leading the twist, as illustrated at the start of Figure 4.2. The binary representation of the hashtags reveals their relative hierarchy.

Every twist contains the tags found along a random path of the group tree. Such a path is highlighted in Figure 4.1, for instance. In the figure, client `@A` is submitting a message to all subgroups containing leaf subgroup `#T09`, so the tweet must mention `#T09 #T04 #T01 #T00`, thus reaching any client streaming on any of those tags (including `@B` on `#T01`), thus simulating hierarchical groups. We reserve 24 characters in tweets to represent paths, which allows encoding 6 levels for groups with name length two[1], e.g., "#♠0␣ ... #♠5␣ `<message>`". The low-order bits of each tag on the path encode the subgroup id (an integer ranging between 0 and $2^{level} - 2$). In the space of valid names we have defined, there are more than 2^{21} 2-character group names possible. Longer group names are also possible, but fewer levels can be listed in the reserved 24 unicode characters.

Messages composed by the user are appended to a submission queue called an outbox. There is only one such queue per client. To conceal recipients of

[1] multiple hashtags must be separated by whitespace inside a tweet

messages, path selection is done at random, independent of messages waiting in the outbox. If a client were to favour a particular path over any other when posting, an adversary could learn contact information over time. To prevent correlating paths chosen with recipients, our strategy is to generate cover traffic by posting at regular epochs, each time picking a random path in the tree. If a message waiting in the outbox has a recipient on that path, it is elected for submission. Otherwise, a noise twist is posted.

A tradeoff between network bandwidth and message latency then becomes apparent. Subgroups one level closer to the root are twice as likely to be covered by a random path and so, senders will require half the number of sending epochs on average to send something there. However, being one level higher implies streaming about twice the amount of incoming messages. An asymmetry in communication also cascades from this: clients with limited resources can message clients close to the root quickly (e.g., to a large organization), but they have to wait longer to receive a reply.

4.3.4 Encrypting Twists

DistoRt users can encrypt short messages into twists, to one recipient addressed by the destination Twitter handle. Although it would be feasible to exchange shared keys to support some form of group chat, we leave this exploration to future work.

Twists are posted by the client as tweets to the user's own public feed. The twist contains a list of hashtags on its envelope targeting the destination anonymity group (as described in Section 4.3.3). The binary body of the twist is unicode-encoded, to respect the 140-Unicode NFC character limit of tweets. We use base16k encoding, which encodes 14-bits of binary data into a single unicode code point[2]. It is not the most optimal bit packing, but it is simple and renders as printable natural (human) language characters. The high-level syntax of a twist is shown in Figure 4.3. Words in angled brackets in the following text, e.g., `<pln_t>`, refer to nodes of the figure.

The twist can only be submitted to the network after the client has learned a valid certificate for the recipient. Then, the client will form the plaintext of the twist by encoding the user's message in utf-8 (`<m_utf8>`), prefixed with its length

[2] i.e., within a contiguous region of $16384 = 16k = 2^{14}$ code points of the basic multilingual plane that is free of combining, reserved, or unallocated characters and surrogates.

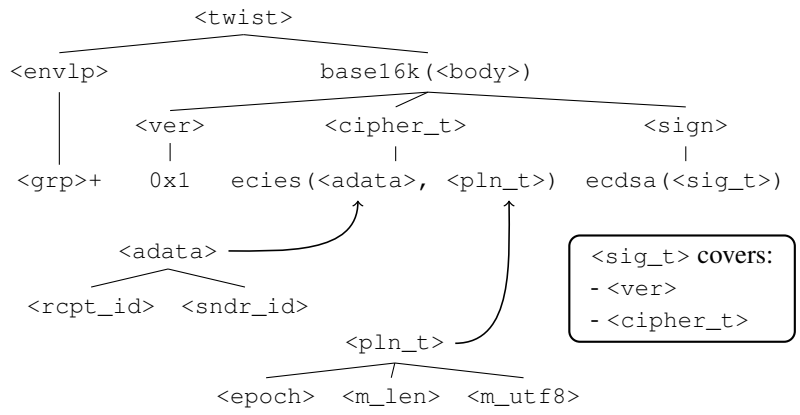

Figure 4.3: Message structure of a twist, encoded in the payload of a tweet. The text composed by the user is encoded into `<m_len>` and `<m_utf8>` (in the lower tree). Other fields are computed by the client at the time of submission.

(`<m_len>`), and then pad it to a fixed length.

We encrypt messages using a hybrid scheme based on ECIES-KEM [59]. The asymmetric part of the algorithm derives a shared secret from the public encryption key of the recipient (a point (x,y) on the elliptic curve) exponentiated to a random number r. The key-derivation function derives a 256bit AES key to encrypt `<pln_t>` and additional authenticated data `<adata>`.

We use AES in the counter mode with CBC-MAC (AES-CCM) [72], configured with a 64bit authentication tag, and a fixed IV. Because each message uses a different random key, we fixate the IV to save space in the transmitted message. This cipher mode supports passing in associated authenticated data, `<adata>`, to be included in the integrity check, but excluded from the symmetric ciphertext output. We include in the checked data: the recipient's 64bit Twitter ID `<rcpt_id>` and the 64bit sender ID `<sndr_id>`. In the plaintext, we also include a 32bit monotonically increasing sequence number based on time, `<epoch>`. This number corresponds to the absolute time at which the message is posted, rounded down to the nearest 256s (4min 16s) boundary. This rough measure of time amortizes time delays in submission, reception, and decryption between the sender, receiver, and Twitter.

The output ciphertext `<cipher_t>` consists of the concatenation of the (ran-

dom) elliptic curve point and the symmetric ciphertext. The scheme presented above allows users to draft messages composed of up to 93B of utf8 text, per twist.

SECURITY. In a multi-user setting, we must prevent malicious users from re-submitting (possibly tampered) messages from other users. The authenticated-encryption mode AES-CCM [72] provides non-malleability [33]. In our case, it prevents undetected tampering of all contents from `<adata>`, and `<pln_t>`: the message sequence, the sender, the receiver, the user message length, and the user message content. Folding in the ID of the sender prevents message replay by other users. Folding in the sequence number prevents Twitter from unreasonably delaying or reordering messages.

The key-encapsulation mechanism ECIES-KEM can be proven to be secure against adaptive Chosen Ciphertext Attack (CCA) (non-malleable), by using the random oracle assumption [10, 13] combined with the assumption that the Computational Diffie–Hellman (CDH) problem is hard, even when given access to an oracle for the Decisional Diffie–Hellman (DDH) problem [58]. The latter assumption is called the gap-CDH assumption. The intractability of the gap-CDH is a reasonable assumption to make in prime multiplicative groups, such as in the chosen P192 curve [53].

Another necessary property in our setting is key-privacy [11]. Roughly speaking, it implies that an eavesdropper learning a new ciphertext does not gain an advantage to infer which public key was used to encrypt it. ElGamal in a group of prime order is proven to have the desired property under the hardness assumption of the DDH problem [11][3]. The proof for the key encapsulation we use follows similarly.

Every client will attempt to decrypt every twist it receives. With probability 100%, a client will successfully decrypt twists intended for it. Tampering with the ciphertext or the message sender (e.g. if Twitter spoofs messages, or if other clients replay another client's twists) will cause the receiver to fail the twist's decryption, with very high probability. Similarly, it is only with very small probability that a client will successfully authenticate a message not actually intended for it.

[3] the proof is in the full-version of their paper, to which we provide a link in our bibliography

Signing Twists

We wish to prevent Twitter and/or Twitter apps with write permission to the user's account (e.g., with OAuth), to spoof the client's twists. To do so, the client signs twists with ECDSA using the sender's current signing key. The signature (`<sign>`) authenticates the ciphertext (`<cipher_t>`), not the plaintext. The signature also covers other fields that should not be spoofable, such as protocol version (`<ver>`).

The presence of the sender ID in the authenticated data of the symmetric ciphertext prevents an adversary (or malicious user) from successful replay by stripping the signature from an existing twist, and appending its own signature.

Threat of Public Record

We are cognisant that Twitter, for all intents and purposes, is functioning as a public record. Tweets from many accounts are shared so widely that they are effectively non-repudiatable.

We consider here the following scenario. The intended recipient of a twist widely exposes the plaintext of the twist as well as the twist itself, which includes the handle of the sender. The sender is not likely to be able to deny that he is the author of the twist. We provide a heuristic approach to a future deniability extension for our scheme.

The symmetric encryption we chose is efficient and provides semantic security, but its non-malleability makes it hard to disprove that a certain plaintext produced the ciphertext on public record, once the key and parameters of the encryption are revealed.

A scheme that encapsulates ElGamal encryptions inside the hybrid encryption scheme detailed above may provide key-privacy, non-malleability, and deniability. Suppose that the symmetric plaintext (`<pln_t>`) embeds multiple ElGamal ciphertext blocks of the form $(R,C) = (g^r, m \cdot g^{rs})$, where r is a randomly generated (or derived) exponent, g is the generator of the group, m is an embedded user message encoded as a group element, and g^s is the public key of the receiver.

Suppose also that the recipient with public key g^s exposes a message's plaintext m by publishing the symmetric key and $P = g^{rs}$ from the embedded ElGamal ciphertext on Twitter, along with m.

To refute having sent m, the sender randomly picks an innocuous alternate message m' according to some distribution with high entropy (e.g., by picking a random phrase of the right length out of War and Peace [135]) and computes P' such that $m' \cdot P' = m \cdot P = C$. The sender can then challenge others to distinguish between $(R,P) = (g^r, g^{rs})$ and (g^r, P'), where P' inherits a high entropy distribution from m'. This is similar but not identical to the Decisional Diffie–Hellman problem.

To undeniably expose a sender in this messaging mode, the recipient could reveal its private key s. However, revealing s would equally expose all of the messages encrypted for the recipient with that key – making it a very unpalatable and possibly-damaging disclosure.

We leave as future work work clarifying the formal deniability of the above, and the development of a scheme that provides all of key-privacy, non-malleability, and deniability.

4.3.5 Cover Traffic

To mask the messaging distribution of users, and prevent a passive adversary from correlating recipients' message requests and replies (which would greatly reduce the effective receiver anonymity), the sending process in the client follows a deterministic submission schedule, independent of the set of online users currently participating.

We fix the sending rate of each client to one message every 15 minutes, plus or minus some error – or roughly 96 messages per day. Comparatively, the group of heaviest American text users sent on average 87.7 messages per day, according to a 2011 study [111]. This rate will have to accommodate most users initially. A higher frequency lowers the expected wait time for messages in the outbox, but naturally, imposes higher network and computation (decryption) overhead on other users.

Recall from Section 4.3.3 that the client generates a noise message when it cannot find an eligible message in the outbox. Noise messages are encrypted with the client's own public key, but they have otherwise a format and length identical to signal messages. So, noise messages are, outside the generating client, indistinguishable from signal messages.

4.3.6 Receiving Twists / Streaming

Twitter streams filter tweets containing hashtags for the client's selected subgroups. This list of hashtags is reflected in the certificate posted by the client. Each incoming twist is decrypted to determine if the client is the intended recipient, by performing an integrity check over the AES-CCM MAC. A matching MAC requires that the correct shared key, authenticated data fields (`<adata>`), and message ciphertext (`<cipher_t>`) be recovered as they were during the encryption. A receiving client attempting to decrypt a message intended for another would derive a different symmetric key, for instance.

The authenticated fields are retrieved directly from the Twitter metadata accompanying a received tweet. The sequence number field (`<epoch>`) is part of the plaintext, but requires special attention. Because the tweet timestamp is not yet available to the sender at the time the message is encrypted, the authentication covers instead a truncated value of the sending client's wallclock time. This is recomputed on the receiver's side using the timestamp reported in the twitter metadata, and compared with the one found in the decrypted plaintext. If the two values drift by more than one epoch, the integrity check will fail. Allowing a delta of 1 covers boundary cases where a tweet is encrypted at the edge of the epoch.

The algorithm to process incoming messages is straightforward. A client attempts to decrypt every incoming message, in a best-effort manner. Messages that fail the integrity checks during decryption are discarded. Those that decrypt successfully are then, in a second pass, authenticated with the signature accompanying the message. The signature verification requires the public signing key of the sender – if the client is unaware of the public key at the time the twist is received, the verification cannot proceed. Signature verification happens after decryption, but the signature verification and decryption could in principle be done in parallel if there were compute resources to spare.

4.4 Evaluation

We wish to evaluate the practicality and scalability of DistoRt's messaging capabilities. With benchmarks, we establish the relation between resource usage (CPU and network) of our prototype application, and the anonymity group size in which

a client participates. From this relation, we report on the maximum group size and messaging frequency which allow DistoRt to remain practical for a client, given consumer-grade hardware and network connectivity, and Twitter's ToS.

We have implemented our client in a Chrome browser extension to more easily interact with Twitter and GitHub's web APIs, and to facilitate installation. It consists of a combined total of approximately 9000 lines of JavaScript, CSS, and HTML. Our cryptographic primitives are prototyped with a modified Stanford Javascript Crypto Library (SJCL) [121]. Our patch to the library is around 300 lines of code, consisting mainly of encoding utilities for representing and compressing elliptic curve points differently.

4.4.1 Maximum Rate of Incoming Twists

We first determine the maximum group size that could be supported, given unconstrained network bandwidth. We say a group is *root-supported* by a given client, if the client can decrypt twists at a faster rate than their rate of arrival on the network, when listening on the root subgroup. We note that this is a conservative metric, as a client able to process the twist traffic of a group where all members are at the root, should be able to take part in groups that are powers-of-two larger by moving to a subgroup.

We have instrumented the client to measure the time taken for a single CPU core to decrypt and decode noise twists in a micro-benchmark. Decoding noise twists represents the normal case scenario, where most twists received are not intended for the receiver (so they fail the MAC check), and are dropped without performing a signature verification. Only signal twists intended for the client will involve a signature verification. The results are shown in Table 4.2, taking the median of 10 runs of 100 iterations of the decryption and decoding routine.

Given a group where each participant posts once every 15 minutes (900 seconds), a client at the root of the group will have to process on average $\frac{1}{900}$ twist/s from each participant. Consequently, the theoretical maximum root-supported group size of a client is approximately 900 times its maximum decryption rate: a client able to decrypt at a rate of 50 twist/s can support a group size of approximately 45,000 users, and every rate increase of 10 twist/s translates into being able

Table 4.2: Processing rates of DistoRt messages (10 runs of N=100 encrypted messages, 1 core, showing median).

Processor	Environment	Rate (twist/s)
Intel Core i7-7500U 2.7GHz	Chrome 60 Windows 10 Pro v1703	89.9
Intel Core i7-4870HQ 2.5GHz	Chrome 59 Mac OS 10.12.1	80.4
Intel Core i5-750 2.67GHz	Chrome 58 Ubuntu 16.04	67.9
Apple A7 1.3GHz	Safari iOS 10.3.1 iPhone 5s	18.3
Apple A6 1.3GHz	Safari iOS 10.3.2 iPhone 5	10.2

to support a group that is 9,000 users larger. We find results from Table 4.2 encouraging, as one core from a Core i5 750 (a product released in 2009) can root-support (in this 15-minute setting) groups composed of over 61,000 users, assuming an accommodating network connection.

We have measured the maximum decryption rates of various clients, in Chrome on desktop computers and Safari on mobile. To run the experiment on Safari for mobile, which does not support Chrome extensions, we have extracted the platform-independent code responsible for processing twists and served it to the browser via a static webpage. We surmise the results we obtained would resemble those from an installed/native HTML5-based application performing similar operations. Devices with less compute resources can still take part in large groups, but they might need to adapt their listening position in the group accordingly.

The benchmark used to produce Table 4.2 processes incoming twists in batches through the main JavaScript thread in the extension. As such, we note that it only exercises a single core. In practice, since twists can be decrypted independently from one another, the process could take advantage of parallelism (e.g., by scheduling web workers) to improve the figures presented. Moving from JavaScript crypto to native crypto routines would also be a natural next step in achieving a (linear) improvement over our prototype, but this is still to be explored.

4.4.2 Maximum Client Bandwidth

We wish to evaluate the network requirements of running DistoRt in large groups. Twitter and GitHub require the manual creation of real accounts (with associated phone numbers). As such, to evaluate the bandwidth requirements we artificially reproduce the traffic of a larger group using a limited number of real accounts. We configure each client to post 1 twist per minute, or 15 times the normal rate, limited by Twitter's ToS-advertised maximum posting rate of 2400 tweets per day.

Fortunately, the network resource usage can be predicted as a function of the number of users in the group and the listening position of the client. Bounded by the logistical limits of creating accounts, we have verified that the predictions hold for traffic equivalent to 240 normal users in a group, posting at 15 minute intervals.

We measured the traffic received through the Twitter stream as reported by Chrome's developer tools. We use these values to generate Table 4.3. Averaging over 1000 received twists, we calculate an average receive cost per twist of 1.30KiB. We use this value to derive the daily receive cost based on group size and level. Listening in a subgroup 1 level lower divides the RX traffic in two. The cost for sending is constant at 3.3KiB per twist, regardless of group size and group level, and is added to the receive cost to form a total. We project values for higher group sizes.

The numbers presented in Table 4.3 include only the cost of sending and receiving twists, not certificates. Traffic from certificates is minimal, as it represents a small 0.3% increase in twist traffic over the figures from the table (1 additional tweet per user every three days). Each certificate verification incurs a cost of 1.2KiB to compare with the copy on GitHub. Similarly, the network cost of verifying your own certificate each hour amounts to a small 1.15MiB/day.

From the results in Table 4.3, we can infer that a user whose client receives 50 tweets per second will witness a daily network usage of approximately 5.5GiB. This is what one would perceive at the root of a group of 45,000 users (or alternatively, at the second level of a group of 90,000 users, and so on). This figure is comparable to 2 hours of streaming Netflix HD content [106].

Table 4.3: Network usage of one client as the size of the group and the subgroup (listening position) varies.

	*Network Usage (MiB/Day)	
Group size	Level 0 (Root)	Level 1
10	1.52	0.91
100	12.5	6.4
1000	122	61.3
10000	1220	610
45000	5490	2750

* *projected on 1.3KiB RX and 3.3KiB TX costs*

4.4.3 Twitter API Limits

The DistoRt client is also subject to the limits published in the Twitter terms of service. We assume that the vast majority of clients will operate DistoRt using a freemium (normal) Twitter account with normal privileges. On such accounts, Twitter enforces imposes rate-limits for most operations in their API, namely for sending and streaming messages. These limits have implications on the size of the anonymity groups in which DistoRt users can feasibly participate.

The rate limits of the freely-available streaming API, to our knowledge, are not officially published by Twitter. However, it is reported that Twitter limits the streaming API to 1% of the "firehose" [52], or in other words 1% of all tweets currently being tweeted. This limit is applied per user access token [125]. Furthermore, if a client cannot process all the tweets pushed on the stream, Twitter will signal this event to the client. Unfortunately, Twitter does not indicate which tweets are dropped [125].

Twitter can process over 500 million tweets per day [127]; these volumes fluctuate leading to commensurate reductions in the number of tweets available through the streaming API, for example there is a reduction in coverage during Western holidays [52]. From this figure, we can conservatively estimate a volume of 5000 tweets sent per second. Such rates would enable a DistoRt client at the root level of a subgroup to receive approximately 50 twists per second, which represents 45,000 active DistoRt users sending on regular 15 minute intervals. As we show in our

benchmarking results (Section 4.4.1), mid-range consumer hardware can support a decryption rate of 50 twists per second.

As every stream maintains a persistent socket to Twitter, if all 45,000 users were at the root level, Twitter would see total network traffic of approximately 266TB per day, or 8PB per month pertaining to DistoRt. Based on the measured internet traffic mixture reported by Sandvine, and Cisco's reporting on all IP traffic, we estimate Twitter's current total network usage to be on the order of 400PB per month [86, 117, 118]. Thus, in the absolute worst case, we would expect a group of 45,000 DistoRt users to account for less than 2% of Twitter's traffic.

The Twitter API unfortunately provides only limited efficient retrieval of non-live tweets. This shortcoming forces our clients to remain online if they wish to reliably receive messages. Two potential solutions we have considered to alleviate this requirement are:

1. Serving compressed daily "twist archives" from an external mirror site or torrents. This would allow clients that were disconnected from Twitter for a prolonged period to download missing archives.

2. Forcing clients to submit new twists as "replies/retweets" of n-day-older twists. Twitter inlines two more tweets of a reply-chain in the JSON tweets served by the streaming API. Taking advantage of tweet inlining can increase reliability, and reduce some of the metadata and posting overhead.

4.5 Related Work

Mapping our work onto the classification axes laid by a recent survey on anonymous systems [65, 66] would yield a unique contender in the space. One of our principal objectives was to create an anonymous messaging system that required no new infrastructure: DistoRt is a broadcast-based approached to anonymous messaging built on deployed internet services. It requires no manual work in *trust establishment*. Namely, it requires no trust in a single provider authority. And even though it is overlaid on services which serve data publicly, it provides *conversation security* (protects conversations), and aims to solve the *transport privacy* problem (hide metadata).

Encrypted email systems, such as PGP and S/Mime, are perhaps the first examples of usable modern solutions to send messages over insecure channels. They are still in use today, albeit mostly by privacy advocates. They can provide end-to-end confidentiality, but their main drawbacks are key distribution: keys can be spoofed, and confusing user interfaces make web-of-trust key distribution only work for the most resolute enthusiasts [73].

End-to-end encryption, even in absence of technical flaws, only solves the privacy problem partly however. Namely, it generally provides confidentiality of the content of messages, but not necessarily of conversation metadata, such as message times and contacts.

The Signal messaging protocol (and associated double-ratchet family) [16, 115, 116] is currently put to use by several mobile applications, namely by WhatsApp, the Signal app, and Facebook Messenger. It protects long-running conversations by rotating keys, but leaves the task of protecting conversations (who, when, how) to the messaging service provider. WhatsApp, for instance, disappoints by logging user interactions, and polling user contacts [114].

End-to-end mobile crypto is shiningly alluring to users, but unless protocols and application code can be inspected and audited, it is difficult to assure its correctness. Mobile applications generally permit ease-of-use and ease-of-adoption, but it remains that an unprecedented greater amount of trust has to be placed on the software and hardware platforms of the devices.

KEY DIRECTORIES. Trust establishment is the process by which a participant first acquires and then authenticates keys of another user. In our work, we assume users know *a priori* the usernames of the users they wish to talk to. We distribute keys on Twitter and GitHub, which act as our key directory. We impose as a requirement that privacy be preserved when retrieving keys (e.g., by passively listening, or with cover traffic), and that there be a way to authenticate keys when both directories or their operators can be malicious (but non-colluding).

ECT (extended certificate transparency) [56] (and to some degree CT [36]) allow end-user certificate distribution in the context of email. It allows email providers to vouch for their users public keys. CONIKS [44] is another distributed key directory. CONIKS offers some privacy advantages over ECT, namely that it

prevents enumeration of usernames from the directory. Their goal is transparency, i.e., to allow users (or auditors) to verify that keys are stored correctly, and detect equivocation. These systems however offer no privacy protection when querying users keys – the key exchanges are performed via central directories.

Keybase [104] is another transparent public key directory service deployment, but with a social network flavor, closer to our approach. Users create a Keybase account in the Keybase directory, to which they upload their public keys. Keybase users can then associate their Keybase account to a set of existing external online social networks (OSNs) accounts. This is done by uploading proofs of Keybase key ownership on the external accounts. Keybase trust is based on Keybase usernames, which means that at the point a key is imported, associations between a keybase username and the external accounts must be verified manually. Once a key is imported and "trusted", clients will sign web-of-trust-like "statements" to provide reputation hints to other users, but the out-of-band verification is still required.

We focus on reusing infrastructure and using only the identities that users already have. With DistoRt, there is no need to trust a middleman, and this allows us to automate key distribution completely. We posit it is better to start the chain of trust with an established Twitter handle, than with a Keybase handle plus another social account of the user's choosing. Signing previously-accepted keys does provide the convenience of skipping redundant checks across devices, but we consider this to be a privacy violation with respect to our trust model and design goals. All of the three directory approaches require an external auditor to prevent fork-style attacks on the directory.

Stadium [64], and the related Vuvuzela [67] system, consists of scalable mixnets that achieve provable guarantees of anonymity by bounding metadata leakage with differential privacy. A successful deployment of Stadium (or Vuvuzela) would require participation from multiple collaborating organizations, which is not a small feat. Furthermore, they do not provide key distribution. Stadium suggests that Alpenhorn [37] be used to distribute keys, which uses IBE to establish keys. To be more specific, it allows users to communicate without knowing each other's public keys. However, this method of key generation, IBE, requires that a master key be held on a central service, which implies that users must trust the service not to engage in spoofing.

The identity directory providers above show promise, but have in practice, so far, failed to gain wide deployment outside technically savvy groups. Wide-scale deployment of these systems would require new organizations to create, maintain, and support networks of new identity providers, and for users to recognize new public/social identities.

PRIVATE INFORMATION RETRIEVAL. PIR offers an interesting mechanism in that it allows clients to retrieve information from a database without revealing to the servers which particular item was read. Pung [4] provides metadata privacy, and uses a PIR database service for message storage. Messages are retrieved from the database using labels that are derived from a shared secret established between two users. Riposte [17] uses PIR in reverse, that is, by hiding which database row was written instead of read. These PIR databases are not currently deployed at scale and also suffer from very high network costs at small cluster sizes. In addition, these systems do not provide user authentication based on publicly-recognizable identities, and they do not support in-band distribution of public keys on their own, which exposes users to traffic analysis on public key retrieval patterns.

ROUTING THROUGH EXISTING SOCIAL GRAPHS. Drac [19] and Pisces [50] mix anonymous messages through an OSN backbone. A primary challenge with this approach, like for Tor, is that the circuits must be formed carefully to prevent an attacker from controlling the flow of messages through the network. Dynamix [51] improves on the circuit selection by compensating for nodes joining and leaving the graph. In general, especially in the case where messages are routed through OSNs, adding hops increases latency dramatically – but using a proven scalable service like Twitter as a broadcast is essentially a single hop. Key distribution is provided out-of-band for all these systems, which is impractical.

VirtualFriendship [8] protects online activity of users in Facebook by routing site activity through Tor circuits. It is a peer-to-peer Tor overlay on top of the OSN. It requires requests to be authenticated with group keys distributed out of band. Similarly, a report on Blindspot [28], describes an onion-routing design overlaid on top of existing OSN infrastructure. Their design achieves high-latency anonymous communication (on the order of a day) by routing messages over friend-links, with probabilistic unobservability. Unlike VirtualFriendship, it does not require public

keys to be exchanged out-of-band, but routing of messages can be influenced by malicious nodes in the OSN.

AnonPubSub [22] presents a set of peer-to-peer (P2P) overlay techniques to anonymize publishers and subscribers' interests. Contrary to our approach, they avoid having a central broker, and distribute brokering onto all participants. We use cryptography to prevent Twitter (and all clients) from linking senders and receivers, whereas they have to use a variety of strategies to reconfigure the routing or network to prevent collusion from multiple clients. Their key management is not immediately deployable, as it requires a trusted server to sign client keys. Nodes also need to exchange keys, but the key-agreement mechanism is not described in depth. In a different publication [20], it is mentioned that key distribution can be achieved with two (new) trusted third parties, but the anonymity of the system hinges on them not sharing information.

TWITTER OVERLAYS. Twitterize [21] is an android application which disseminates sensor data on Twitter anonymously. It does so by using the follower graph similarly to a mix, relying on followers to relay messages. No extra service outside Twitter is necessary to communicate, but secrets must be exchanged out-of-band. Also, every client stores every symmetric key, so compromise of a single device could expose all messages (which are only AES-encrypted). The system also relies on clients to forward messages in the P2P overlay, which makes it difficult to bootstrap.

Twistor [2] also creates a P2P onion routing overlay on Twitter, with goals similar to ours. However, their solution is impractical, because it relies trusting a new central service to store client keys. Also, like Twitterize, they rely on participating nodes within follower edges to route messages. This makes the system unusable until there is a sufficient number of participating nodes in the graph, and requires additional trust from nodes in the network. Keys in Twistor are not shared, unlike in Twitterize, which implies that clients cannot determine already-received message duplicates from new messages, opening the way to a flooding vulnerability.

Hummingbird [18], a privacy-enhancing service which hides relationships between Twitter followers, provides confidentiality for content and anonymizes hashtag subscriptions. A major downside to their solution is its introduction of a new

central (Twitter-like) service to relay tweets and enforce ACLs before submission to Twitter. It therefore doesn't meet our needs, and their threat model does not cover equivocation. #hoot [5] modifies Twitter clients to specifically protect against censorship rather than to provide confidentiality or anonymity.

ANONYMITY WITH BROADCAST. Other approaches employ broadcast to provide receiver anonymity. Hordes [40] anticipated the adoption of IP-level multicast routers to increase anonymity of senders. Unfortunately, IP-level multicast is not widely deployed. Twitter's availability, on the other hand, is widespread[4]. We could also potentially substitute Twitter for another OSN offering a publish/subscribe model – whereas it is harder to replace a technology requiring worldwide deployment of new hardware. Hordes also does not provide in-band key distribution.

A multicast tree layer can be constructed at the application-level, over a P2P network. One approach in particular uses such trees to allow any node to broadcast to all other peers, but leaves the problems of bootstrapping the list of users and passing their identities/keys to future work [70]. These approaches make it difficult to make assumptions about the quality of selected tree nodes to route messages.

Some protect the anonymity of receivers by combining multicast with multiple identities for each receiver [71]. The threat model there is different than the one we present; the identity of the receiver is meant to be protected from all senders as well. We require that key certificates distributed via multicast be authenticated/genuine, which would go against their incomparable key system.

Other approaches rely on a mixture of p2p-routing to anonymize the sender and multicast subscription trees to propagate replies back to the source [48, 55]. Our approach does not require a random route to be selected on the forward path, as we consider the property of having unlinkability with the receiver to sufficient given our threat model. P5 [57] provides sender and receiver anonymity by creating a hierarchy of broadcast channels, along with a hierarchy of mixes in the process. We believe the need for mixes, as well as the requirement for out-of-band key distribution, will make its deployment unlikely.

Finally, there is Bitmessage [82], which performs broadcasts by submitting

[4] in most geopolitical spheres

proofs of work into a global blockchain. Public key hashes need to be exchanged out-of-band, and it would seem underlying public keys must be retrieved explicitly via an API call (a formal document about the full protocol is missing). They describe a mechanism by which their global stream (chain) can split into two smaller streams (each with its own blockchain), and nodes are responsible to maintain lists of all nodes in their two child streams. Nodes seeking to communicate with another node must actively query progressively-smaller groups to obtain lists of peers to find its stream. Side-channel leakage implications of their peer address resolution mechanism are not formally stated, nor are the provisions to prevent users from arbitrarily reporting incorrect peer lists. The blockchain approach, however, does have the propensity for an interesting model where a messaging system can be financed through micro-payments, and can present a technical obstacle to spamming.

4.6 Discussion and Future Work

WHAT DOES IT MEAN TO EQUIVOCATE? A service provider equivocates by publishing alternate versions of content to different users, usually in a hard to detect manner.

We consider two opportunities for equivocation: on message content, and on certificate posts. The hosting provider for messages, Twitter, has little opportunity to equivocate on the content of messages, because of integrity checks (signature, and CRC). There is however a small window of opportunity to alter the time of submission of messages present in the metadata, forward of backward. Adjusting the time by an amount larger than the epoch granularity would amount to denial of service, as it would invalidate the signature. Altering it only slightly could cause messages to be reordered differently. But, conducting an effective reordering attack would require prior information on the nature of the messages and their recipients. So, we do not see an incentive for the message hosting provider, Twitter, to do so.

The adversary could try to equivocate on public keys in an attempt to intercept the victim's communications. This would need to be done without leaving a trace, as evidence of this kind of attack would tarnish a provider's honest reputation. But, a certificate is only usable if its copies on the two certificate providers are identical.

If the two copies differ, then the post will be ignored, and this will amount to DoS for that user until the issue is resolved. Additionally, if the client verifies its own cert and finds two different copies, the user is alerted. Unfortunately, the client cannot automatically determine the underlying cause for a mismatch: it could be due to a client reconfiguration (e.g. factory-reset of the client), to an account compromise (e.g., stolen password), or to the user deleting an old post manually.

To mount an effective certificate spoofing attack, both providers would need to collude and present the same certificate for a victim user. One effect of a successful spoofing attack would be to "redirect" all or part of the incoming correspondence of the victim to the attacker. To spoof a user's outgoing traffic would require also equivocating on the sender's certificate. This form of collusion is outside our threat model. It would require external auditors or a non-repudiable log of certificates to defeat all possible equivocation attacks by the providers.

ACCOUNT COMPROMISE. The damage caused by an account compromise of only one of Twitter or GitHub is limited to Denial of Service (DOS) for the victim. If both accounts are compromised by the same attacker, then the certificate can be spoofed, and the client MITM'ed. In the absence of equivocation however, the DistoRt client will eventually learn about the change, and the situation can be remedied.

Chaining signatures in certificates would prevent an attacker with write access to both accounts to spoof a new certificate, because doing so requires knowledge of the previous private key. Still, chains are more expensive to verify than a single certificate, and the security gain is limited. This approach would also not fully protect against provider equivocation, because colluding providers could still "lie" about a complete chain instead of just one certificate.

If an attacker steals the keys from the client, that attacker can assume the user's identity for receiving and posting. Posting on behalf on the victim would be careless, because the public visibility of messages would immediately expose the attack. Passively eavesdropping on client communications is a more prudent attack, but is thwarted by the next key rotation.

The client does not store passwords to the social accounts, only temporary/revocable credentials. Consequently, to revoke access to a stolen computer (contain-

ing a configured client, keys, and credentials), for instance, a victim user can simply change account passwords, clear OAuth credentials, and post new keys to revoke the compromised ones. A special broadcast could be added, to notify correspondents possibly affected by the compromise. To regain control of a social account in case its passwords are also compromised, the victim must go the support route.

GROUP CREATION & GROUP PARAMETERS. There is a potential danger into joining a group that is not yet popular, in that it may reveal a certain interest in communicating with small anonymity set to an observer. This is perhaps less applicable to the initial default group that all clients join, because it signals more an intention of using DistoRt than necessarily talking with users in that group.

There is no explicit group creation operation. The client joins a group by including the associated tag in a field of the certificate. Streaming for the `#dtcrt` tag allows learning new groups. One can also use this mechanism to "sense" the current number of users in the group by monitoring other certs carrying the same tag. We suspect that over time, knowledge of popular groups would circulate within the DistoRt community.

As groups grow, users no longer able to process group traffic might be forced to migrate towards a smaller group. This is a loss for the users still remaining in the group. To support groups that can accommodate larger amounts of smaller nodes over time, we have considered reserving a few bits of the group name to encode a multiplier over the default sending rate of 15 minutes. This would allow per-group sending rates.

SYBILS AND FEED POLLUTION. An adversary could mount a Sybil attack on a group to create the appearance of a larger anonymity set than the effective anonymity set. This attack remains expensive as it requires a large number of bots to facilitate, and in the end, lowers only the *apparent* anonymity of the group, i.e., the anonymity of the users not in the Sybil set, remains unscathed.

Sybils remain a serious concern for Twitter. Some estimate that up to 15% of active accounts may consist of bots [68]. Furthermore, large bot-nets continue to be discovered [24], despite extensive research into Sybil detection. This speaks to the inherent endogeneity of the problem; a good Sybil detector can be used to vet bots and therefore create Sybils that are tougher to detect [1, 3, 68, 75, 76].

Fortunately, "legitimate" Twitter accounts used for DistoRt, will benefit from improvements to Sybil-detection techniques. The work by Winter *et al.* [75], in particular, showed considerable success using other network level markers such as IP address, uptime, geographical distribution, and OS and browser version as well as patch versions of the OS to detect Sybils in the Tor network. We hope that similarly, Twitter could apply similar tricks.

To reduce the feed notifications received by Twitter followers of an account using DistoRt, we consider two approaches. One approach to reduce pollution is for the client to delete twists immediately after posting them to Twitter. The twists will make it through the streaming API, but it will allow the user's normal posts to stay at the top of the feed. Another approach is to insert a `@foo` mention at the start of twists, where `foo` is an account with no or very little followers. This would effectively notify only the users in the follower-intersection of the sender and `@foo`.

MOBILE. Our system is admittedly more demanding in network and compute than the now-popular messaging mobile applications. DistoRt might thus seem unsuitable for mobile, at first glance. Nevertheless, we believe that a natural extension to the system would be to allow the client to run remotely from the user. It is technically feasible and realistic to run the client in the cloud (or at home), and notify the mobile user over a secure connection on the arrival of a message. However, this is beyond the scope of the paper and its evaluation.

MESSAGE LENGTH AND CRYPTOGRAPHIC PARAMETERS. Binary encoding allows us to get more out of Twitter's exacting character limits. But, we've considered multiple options to extend user message lengths even further. The options we list are complementary to one another.

One possibility is to stego-encode twists into images in a way that resists Twitter's image post-processing. A benefit from this would be to have each twist carry two messages: the encrypted payload, and a legible message in the graphic itself. It however is a taxing option for the network, as image data has to be retrieved with a separate request external to the tweet stream.

A second option is to chain twists into tweet replies. In Twitter's stream implementation, each incoming tweet is accompanied with up to 2 inlined replies. Thus, it is possible to publish multiple tweets at once to subscribers of the group

without requiring group members to retrieve tweets one by one. We could utilize the network better by omitting group tags (and other fields) from all but one tweet in the chain.

A third option is to compress the ECC points produced in the ECIES-KEM key encapsulation. The y-coordinate of the point on our chosen P192 elliptic curve is a root of the elliptical curve equation $y^2 = x^3 + ax + b \bmod P$, so it can be compressed to a single parity bit (if a root r is odd, then $-r \bmod P$ is even, and vice versa). This allows a reduction of almost half (191/384bits) of the public-key ciphertext, and increases the usable message space (in practice) by 23B. The downside is that it increases the work required for the receiver to determine if they are the intended receiver of a message. We note that approach applies to other curves than P192 as well – in general roots can be found quickly if P is congruent 3 mod 4. Other curves, such as Curve25519 [12] would offer short point representations with very little computation.

A fourth option would be to move the sequence number out of the encrypted plaintext (`<pln_t>` in Figure 4.3) to the authenticated plaintext (`<adata>`). Since it is based on the absolute time, it can in practice be reconstructed on the receiver's side from the creation timestamp reported by Twitter in the metadata. However, in cases where the time of the post is close to an epoch boundary however, a second decryption must be attempted with the bumped sequence number on receipt of the post. The key sboxes can be reused for the second attempt (it is the same key), so it is more expensive, but not necessarily doubly expensive. It is also conceivable to force clients to refrain from sending during boundary periods, but introduces additional complexity. This would save an extra 4B of message space.

After experimenting with different parameters over time, we believe it is important to also mention that our implementation of the protocol has become very modular, and we do not anticipate it would be technically hard to switch to a different curve (with larger orders, e.g., 256bit instead of 192bit), or different encryption parameters offering different security properties.

4.7 Conclusion

DistoRt provides receiver-anonymity and confidential messaging on existing scalable services, Twitter and GitHub, to lower the barrier of entry for new users. DistoRt also offers in-band key distribution, centered around recognizable Twitter handle identities, which drastically reduces the burden of key-management. An advantage of DistoRt over other systems is its simplicity: the security properties of multicast-encryption are easy to reason about. To manage the inherent costs in broadcast, the application provides a mechanism for clients to gracefully adapt their resource usage, without sacrificing effective anonymity. The application is deployable on today's computers and should permit anonymity groups of tens of thousands of users to form organically.

Chapter 5

Conclusion and Future Work

We have shown how disaggregating components of web services responsible for enforcing privacy, and exposing them to clients can both enhance user privacy, and reduce the total TCB of applications. Furthermore, we have demonstrated that our techniques can be applied to new and existing services, without detriment to many of the alluring benefits currently enjoyed from the modern web.

In Micasa, by disaggregating the components of a service responsible for data storage and data authorization, we show that applications can provide additional guarantees about the longevity of data and application functionality. By encouraging developers to declare to which degree functionality in their application depends on their service, application providers can make data and software feature longevity claims that can be verified almost automatically by users. The shift of these component roles from the server towards the client results in added control for the user over their own data, for instance by allowing content to be deleted, while simultaneously allowing the providers to optimize their sites, e.g., via caching.

In Beeswax, we extend the assurance of data and functionality longevity to cover also data confidentiality, *across all* cooperating applications. We argue that the platform increases security. That is, that the platform allows a drastic reduction in the combined TCB of all applications (built with it) to simply that of the platform itself. This benefit was not motivated as such in the Micasa manuscript (Chapter 2), but the platform argument holds equally strongly for both Micasa and Beeswax.

The Beeswax platform minimizes the amount of effort needed to add encryp-

tion to an application. Its API allows developers to explicitly declare access control rules of user data in the application without needing to manage key material and encryption directly: the platform takes care of key distribution and encryption. This results in privacy guarantees that the user can verify. One perceived drawback of performing the cryptography in the trusted platform code, as opposed to from within the application code, is that the application loses unfettered access to the plaintext. To compensate for this loss Beeswax provides "appearance modifiers", which allow the application to modify the look and feel of private data, in a way compatible with modern web application development frameworks.

DistoRt enables users to reason about their anonymity. We present an immediately deployable application for anonymous user-to-user communication. DistoRt addresses problems in existing anonymous systems, namely that multi-hop anonymous networks (as in Tor) make it difficult to make security guarantees, that the reliance on central (certificate or identity) authorities to manage keys is very demanding from a trust point-of-view, and that distributing keys out-of-band is an inconvenience that hinders deployment. Our solution is to lean on the existing services, Twitter and GitHub, for both message delivery, key distribution, and user identity. The result is a conceptually simple and practical anonymous messaging application with end-to-end cryptography. User privacy is enhanced with anonymity guarantees users can reason about, while taking advantage of the current web service ecosystem.

5.1 Future Work

In each of the three main chapters, we have identified potential research areas for the future (Section 2.7, Section 3.7, Section 4.6). But we believe there are also interesting challenges in combining the three approaches together, as we believe the three systems are complementary.

In Micasa, we assume that the storage provider is trusted with the safekeeping of the private user data. We are not concerned about the confidentiality or the possibility of data exfiltration by the storage provider. In Beeswax we allow applications to make guarantees of confidentiality.

We believe a combination of Micasa and Beeswax would be desirable. Users

could control the placement of their own data, and at the same time ensure adequate confidentiality levels. We could imagine, for instance, the combination could ensure that all data being submitted to a third-party storage provider be encrypted using keys satisfying certain access control rules. Since the confidentiality rules would be enforced by the platform, and not the application, one benefit would be to allow migration of user data in a way that preserves pre-established access control rules. A second benefit would be to allow users to delete the authoritative copy of an encrypted file. That can be a useful recourse when cryptographic keys to the file cannot otherwise be revoked.

The key distribution protocol in DistoRt is an improvement over the one in Beeswax, because it protects against equivocation by Twitter against users. Equivocation was not as much of a concern in Beeswax as it was in DistoRt. In Beeswax, the routing for key-agreement messages is performed by the application servers, so Twitter is not in a position to man-in-the-middle. The improved key-distribution mechanism from DistoRt could replace the one in Beeswax. One could also imagine replacing the key-agreement protocol from Beeswax with one that completely works over Twitter. This would allow friendship requests to form even when users are not logged in to the application.

Lastly, the "platform argument" from Beeswax could equally apply to DistoRt. We could build a version of DistoRt that is a platform for anonymous messaging, suitable for multiple applications. Applications wishing to guarantee anonymous communications between its users could embed an API to DistoRt for sending and receiving messages. To display received twists, and compose outgoing twists we could reuse the DOM-isolation technology from Beeswax. This combination has additional challenges however. For instance, displaying a twist inside the application should not leak metadata. A bad implementation would allow a passive application adversary to correlate the time at which twists are received on the network and the time at which plaintext is displayed to the user.

Bibliography

[1] K. S. Adewole, N. B. Anuar, A. Kamsin, K. D. Varathan, and S. A. Razak. Review. *Journal of Network and Computer Applications*, 79(Complete): 41–67, 2017. → page 116

[2] M. S. Alavi. Twistor:anonymous message transfer through Twitter. Master's thesis, University of British Columbia, 2015. URL https://open.library.ubc.ca/cIRcle/collections/24/items/1.0223805. → pages v, 112

[3] L. Alvisi, A. Clement, A. Epasto, S. Lattanzi, and A. Panconesi. Sok: The evolution of sybil defense via social networks. In *2013 IEEE Symposium on Security and Privacy (SP)*, pages 382–396. IEEE, 2013. → page 116

[4] S. Angel and S. Setty. Unobservable Communication over Fully Untrusted Infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, GA, Nov. 2016. USENIX Association. → pages 79, 83, 111

[5] D. Bachrach, C. Nunu, D. S. Wallach, and M. Wright. #h00t: Censorship Resistant Microblogging. *ArXiv e-prints*, Sept. 2011. → page 113

[6] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: an online social network with user-defined privacy. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 135–146, New York, NY, USA, 2009. ACM. → page 40

[7] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting Browsers from Extension Vulnerabilities. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS'2010)*, 2010. → page 51

[8] F. Beato, M. Conti, B. Preneel, and D. Vettore. VirtualFriendship: hiding interactions on online social networks. In *Communications and Network*

Security (CNS), 2014 IEEE Conference on, pages 328–336. IEEE, 2014. → page 111

[9] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *CRYPTO '93*, 1993. → page 67

[10] M. Bellare and P. Rogaway. Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 62–73, 1993. → page 100

[11] M. Bellare, A. Boldyreva, A. Desai, and D. Pointcheval. Key-Privacy in Public-Key Encryption. In *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security Gold Coast: Advances in Cryptology*, pages 566–582, December 2001. The full version with proofs is available on the author's website: https://cseweb.ucsd.edu/~mihir/papers/anonenc.pdf. → pages 80, 87, 100

[12] D. Bernstein. Curve25519: New Diffie–Hellman Speed Records. In *Public Key Cryptography (PKC 2006)*, volume 3958, pages 207–228. 2006. → pages 68, 118

[13] R. Canetti, O. Goldreich, and S. Halevi. The Random Oracle Methodology, Revisited. *J. ACM*, 51(4):557–594, July 2004. → page 100

[14] N. Carlini, A. P. Felt, and D. Wagner. An Evaluation of the Google Chrome Extension Security Architecture. In *USENIX Security 2012*. → page 51

[15] R. Chandra, P. Gupta, and N. Zeldovich. Separating web applications from user data storage with BSTORE. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, pages 1–1, 2010. → page 39

[16] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila. A Formal Security Analysis of the Signal Messaging Protocol. Cryptology ePrint Archive, Report 2016/1013, 2016. http://eprint.iacr.org/2016/1013. → page 109

[17] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An Anonymous Messaging System Handling Millions of Users. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 321–338. IEEE, May 2015. → pages 79, 111

[18] E. D. Cristofaro, C. Soriente, G. Tsudik, and A. Williams. Hummingbird: Privacy at the time of twitter. In *2012 IEEE Symposium on Security and Privacy*, pages 285–299, May 2012. → page 112

[19] G. Danezis, C. Diaz, C. Troncoso, and B. Laurie. Drac: An Architecture for Anonymous Low-Volume Communications. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 202–219. Springer, 2010. → page 111

[20] J. Daubert. *Anonymous Publish-Subscribe Overlays*. PhD thesis, Technische Universität Darmstadt, Darmstadt, 2016. URL http://tuprints.ulb.tu-darmstadt.de/5473/. → page 112

[21] J. Daubert, L. Böck, P. Kikirasy, M. Mühlhäuser, and M. Fischer. Twitterize: Anonymous Micro-blogging. In *2014 IEEE/ACS 11th International Conference on Computer Systems and Applications*. → page 112

[22] J. Daubert, M. Fischer, T. Grube, S. Schiffner, P. Kikiras, and M. Mühlhäuser. AnonPubSub: Anonymous publish-subscribe overlays. *Computer Communications*, 76:42–53, February 2016. → page 112

[23] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-generation Onion Router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, 2004. → pages 4, 79

[24] J. Echeverría and S. Zhou. The 'star wars' botnet with >350k twitter bots. *CoRR*, abs/1701.02405, 2017. URL http://arxiv.org/abs/1701.02405. → page 116

[25] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–, Berkeley, CA, USA, 2010. → page 40

[26] A. J. Feldman, A. Blankstein, M. J. Freedman, and E. W. Felten. Social networking with frientegrity: privacy and integrity with an untrusted provider. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 31–31, 2012. → page 40

[27] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully Abstract Compilation to JavaScript. In *POPL '2013*. → page 76

[28] J. Gardiner and S. Nagaraja. Blindspot: Indistinguishable anonymous communications. *CoRR*, abs/1408.0784, 2014. URL http://arxiv.org/abs/1408.0784. → page 111

[29] R. Geambasu, C. Cheung, A. Moshchuk, S. D. Gribble, and H. M. Levy. Organizing and sharing distributed personal web-service data. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 755–764, New York, NY, USA, 2008. ACM. → page 39

[30] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song. ShadowCrypt: Encrypted Web Applications for Everyone. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1028–1039, 2014. → pages 49, 74

[31] L. Ingram and M. Walfish. TreeHouse: JavaScript sandboxes to helpWeb developers help themselves. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 13–13, 2012. → pages 29, 76

[32] R. Jansen, F. Tschorsch, A. Johnson, and B. Scheuermann. The sniper attack: Anonymously deanonymizing and disabling the Tor network. Technical report, DTIC Document, 2014. → page 80

[33] J. Jonsson. On the Security of CTR + CBC-MAC NIST Modes of Operation – Additional CCM Documentation, 2002. → page 100

[34] M. Krohn, A. Yip, M. Brodsky, R. Morris, M. Walfish, et al. A world wide web without walls. *Proceedings of HotNets-VI, Atlanta, GA*, 2007. → page 39

[35] A. Kwon, M. AlSabah, D. Lazar, M. Dacier, and S. Devadas. Circuit Fingerprinting Attacks: Passive Deanonymization of Tor Hidden Services. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 287–302, Washington, D.C., Aug. 2015. USENIX Association. → page 80

[36] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962, 2013. URL https://rfc-editor.org/rfc/rfc6962.txt. → page 109

[37] D. Lazar and N. Zeldovich. Alpenhorn: Bootstrapping Secure Communication without Leaking Metadata. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 571–586. USENIX Association, 2016. → pages 9, 79, 110

[38] J.-S. Legare, D. T. Meyer, M. Spear, A. Totolici, S. Bainbridge, K. MacRow, R. Sumi, Q. Jung, D. Tjandra, D. Williams-King, W. Aiello, and A. Warfield. Tolerating business failures in hosted applications. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 22:1–22:16, 2013. → page iv

[39] J.-S. Legare, R. Sumi, and W. Aiello. Beeswax: a platform for private web apps. In *Proceedings on Privacy Enhancing Technologies*, PETS'16 (issue 3), pages 24–40, 2016. → pages v, 83

[40] B. N. Levine and C. Shields. Hordes: A Multicast Based Protocol for Anonymity. *J. Comput. Secur.*, 10(3):213–240, September 2002. → pages 81, 113

[41] W. Lueks and I. Goldberg. Sublinear scaling for multi-client private information retrieval. In *International Conference on Financial Cryptography and Data Security*, pages 168–186. Springer, 2015. → page 83

[42] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating JavaScript with Filters, Rewriting, and Wrappers. In *ESORICS'2009*. → pages 63, 76

[43] J. Magazinius, P. H. Phung, and D. Sands. Safe Wrappers and Sane Policies for Self Protecting Javascript. In *NordSec'10*. → pages 63, 64, 76

[44] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman. CONIKS: Bringing Key Transparency to End Users. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 383–398, Washington, D.C., Aug. 2015. USENIX Association. → page 109

[45] L. A. Meyerovich and B. Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *IEEE S&P 2010*, pages 481–496, 2010. → page 76

[46] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, 2010. → page 20

[47] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Safe active content in sanitized JavaScript. Technical report, Google, Inc, June 2008. → page 76

[48] A. Mislove, G. Oberoi, A. Post, C. Reis, P. Druschel, and D. S. Wallach. AP3: Cooperative, Decentralized Anonymous Communication. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, EW 11, 2004. → page 113

[49] P. Mittal, F. G. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg. PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval. In *USENIX Security Symposium*, 2011. → page 83

[50] P. Mittal, M. Wright, and N. Borisov. Pisces: Anonymous communication using social networks. In *Network and Distributed System Security Symposium (NDSS)*, 2013. → page 111

[51] A. Mohaisen and Y. Kim. Dynamix: anonymity on dynamic social structures. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 167–172. ACM, 2013. → page 111

[52] F. Morstatter, J. Pfeffer, H. Liu, and K. M. Carley. Is the sample good enough? comparing data from twitter's streaming api with twitter's firehose. *arXiv preprint arXiv:1306.5204*, 2013. → page 107

[53] T. Okamoto and D. Pointcheval. The Gap-Problems: A New Class of Problems for the Security of Cryptographic Schemes. In *Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography*, PKC'01, pages 104–118, 2001. → page 100

[54] P. H. Phung, D. Sands, and A. Chudnov. Lightweight Self-protecting JavaScript. In *ASIACCS'09*, pages 47–60, 2009. → page 63

[55] M. K. Reiter and A. D. Rubin. Anonymous Web Transactions with Crowds. *Commun. ACM*, 42(2):32–48, Feb. 1999. ISSN 0001-0782. → page 113

[56] M. D. Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *In Network and Distributed System Security Symposium (NDSS). Internet Society*, 2014. → page 109

[57] R. Sherwood, B. Bhattacharjee, and A. Srinivasan. P5: A Protocol for Scalable Anonymous Communication. *J. Comput. Secur.*, 13(6):839–876, Dec. 2005. → page 113

[58] V. Shoup. A proposal for an ISO standard for public key encryption (version 2.1). *IACR E-Print Archive*, 112, 2001. also on the author's site: http://www.shoup.net/papers/iso-2_1.pdf. → pages 80, 100

[59] V. Shoup. FCD 18033-2 Encryption algorithms—Part 2: Asymmetric ciphers, 2004. The official ISO standard can be purchased for a fee, but the Final Committee Draft (FCD) is essentially the same. Author's site: http://www.shoup.net/iso/std6.pdf. → pages 80, 99

[60] E. Stark, M. Hamburg, and D. Boneh. Symmetric Cryptography in Javascript. In *ACSAC '2009*, Dec 2009. → page 68

[61] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières. Protecting Users by Confining JavaScript with COWL. In *OSDI'2014*, pages 131–146, 2014. → page 76

[62] Y. Sun, A. Edmundson, L. Vanbever, O. Li, J. Rexford, M. Chiang, and P. Mittal. RAPTOR: Routing Attacks on Privacy in Tor. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 271–286, Washington, D.C., Aug. 2015. USENIX Association. → page 80

[63] R. R. Toledo, G. Danezis, and I. Goldberg. Lower-Cost ε-Private Information Retrieval. *Proceedings on Privacy Enhancing Technologies*, 2016(4):184–201, 2016. → page 83

[64] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich. Stadium: A Distributed Metadata-Private Messaging System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 423–440. ACM, 2017. → pages 9, 110

[65] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith. SoK: Secure messaging. (extended techreport). Technical report, University of Waterloo, 2005. → page 108

[66] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith. SoK: Secure Messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249, May 2015. → page 108

[67] J. Van Den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 137–152. ACM, 2015. → pages 9, 79, 110

[68] O. Varol, E. Ferrara, C. A. Davis, F. Menczer, and A. Flammini. Online human-bot interactions: Detection, estimation, and characterization. *CoRR*, abs/1703.03107, 2017. URL http://arxiv.org/abs/1703.03107. → page 116

[69] T. Wang and I. Goldberg. On Realistically Attacking Tor with Website Fingerprinting. In *Proceedings on Privacy Enhancing Technologies*, volume 4, 2016. → page 80

[70] Y. Wang and P. Dasgupta. Anonymous communications on the internet. In *Proceedings of Communication, Network, and Information Security*, CNIS'05, 2005. → page 113

[71] B. R. Waters, E. W. Felten, and A. Sahai. Receiver Anonymity via Incomparable Public Keys. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 112–121, 2003. → page 113

[72] D. Whiting, N. Ferguson, and R. Housley. Counter with CBC-MAC (CCM). 2003. Available as RFC3610, and on NIST http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ccm/ccm.pdf. → pages 99, 100

[73] A. Whitten and J. D. Tygar. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, SSYM'99, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association. → page 109

[74] Z. Wilcox-O'Hearn and B. Warner. Tahoe: the least-authority filesystem. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, StorageSS '08, pages 21–26, New York, NY, USA, 2008. ACM. → page 40

[75] P. Winter, R. Ensafi, K. Loesing, and N. Feamster. Identifying and characterizing sybils in the tor network. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1169–1185, Austin, TX, 2016. USENIX Association. → pages 116, 117

[76] Z. Yang, C. Wilson, X. Wang, T. Gao, B. Y. Zhao, and Y. Dai. Uncovering social network sybils in the wild. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 8(1):2, 2014. → page 116

[77] ADSafe Home Page. http://www.adsafe.org. Accessed: Feb 2015. → page 76

[78] Why Can't I Restore My Twitter and Facebook Data? https://web.archive.org/web/20110121003922/http://blog.backupify.com:80/2011/01/12/why-cant-i-restore-my-twitter-and-facebook-data/. Accessed: July 2017. → page 13

[79] Beeswax Platform: Home Page. https://web-priv.github.io/beeswax/. (code for platform and applications). → page 59

[80] Bing Moving to Encrypt Search Traffic by Default. https://blogs.bing.com/webmaster/2015/06/15/bing-moving-to-encrypt-search-traffic-by-default, June 2015. Accessed: Nov 2017. → page 10

[81] Updates to Bing Privacy (IP retention reduced from 18 to 6 months). https://blogs.bing.com/search/2010/01/19/updates-to-bing-privacy-2/, January 2010. Accessed: Nov 2017. → page 10

[82] Bitmessage: A Peer-to-Peer Message Authentication and Delivery System. whitepaper https://bitmessage.org/bitmessage.pdf, 2017. Accessed: April 2017. → page 113

[83] Camlistore Project Home Page. http://camlistore.org/. Accessed: July 2017. → page 40

[84] ChatStep: Secure / Private / Beautiful Online Group Chat. Home Page. https://chatstep.com/. Accessed: July 2017. → pages 7, 44

[85] CIRA Internet Factbook 2016 (data from comScore). https://cira.ca/factbook/domain-industry-data-and-canadian-Internet-trends/internet-use-canada, 2016. Accessed: June 2017. → page 1

[86] Cisco VNI Forecast and Methodology, 2015-2020. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html, June 2016. → page 108

[87] Chrome Developer: Content Scripts. https://developer.chrome.com/extensions/content_scripts. Accessed: June 2017. → page 51

[88] Cryptocat 2: Deployment Notes (move from web to extension model). http://web.archive.org/web/20130609085350/https://blog.crypto.cat/2012/07/cryptocat-2-deployment-notes/, . Accessed: July 2017. At the time Beeswax was written, Cryptocat was still available as

browser extension. It was re-written as a desktop application in 2016. → pages 7, 44

[89] Cryptodog (fork of Cryptocat 2): Home Page. https://crypto.dog/, . Accessed: July 2017. → pages 7, 44

[90] Mozilla Security Blog: Plugging the CSS History Leak. https://blog.mozilla.org/security/2010/03/31/plugging-the-css-history-leak/, March 2010. Accessed: July 2017. → page 62

[91] Diaspora Foundation Home Page (formerly known as the diaspora project). https://diasporafoundation.org/. Accessed: July 2017. → pages 18, 40

[92] J.-C. Sirot. jcsirot/digest.js. https://github.com/jcsirot/digest.js. Accessed: July 2017. → page 38

[93] Mark Zuckerberg Posts: "...the facebook community is now officially 2 billion people!". https://www.facebook.com/zuck/posts/10103831654565331, June 2017. Accessed: June 2017. → page 4

[94] G. Fox. Yahoo Sets the Date of GeoCities' Death. http://www.pcmag.com/article2/0,2817,2350024,00.asp, July 2009. Accessed: July 2017. → page 12

[95] GitHub Help: Name Squatting Policy. https://help.github.com/articles/name-squatting-policy/, 2017. Accessed: April 2017. → page 91

[96] As G Suite gains traction in the enterprise, G Suite's Gmail and consumer Gmail to more closely align. https://https://www.blog.google/products/gmail/g-suite-gains-traction-in-the-enterprise-g-suites-gmail-and-consumer-gmail-to-more-closely-align/, June 2017. Accessed: July 2017. → page 10

[97] Google. Chrome Apps. https://developer.chrome.com/extensions/apps, . Accessed: July 2017. → page 19

[98] Google. What are extensions. https://developer.chrome.com/extensions, . Accessed: July 2017. → page 19

[99] Google. Official Blog: A fall sweep. http://googleblog.blogspot.ca/2011/10/fall-sweep.html, October 2011. Accessed: July 2017. → page 12

[100] Google. Chrome Developers: DOM Attributes now on the prototype chain. https://developers.google.com/web/updates/2015/04/DOM-attributes-now-on-the-prototype-chain, April 2015. Accessed: July 2017. → page 75

[101] S. Lohr. Google to End Health Records Service After It Fails to Attract Users. http://www.nytimes.com/2011/06/25/technology/25health.html, June 2011. Accessed: July 2017. → page 12

[102] Google. Official Blog: A second spring of cleaning. http://googleblog.blogspot.ca/2013/03/a-second-spring-of-cleaning.html, March 2013. Accessed: July 2017. → page 12

[103] Google Hangouts Help. Delete or turn off your message history. https://support.google.com/hangouts/answer/3112001. Accessed: July 2017. Google Chat (now defunct) in GMail had a similar feature, and was replaced on June 26 2017 by Hangouts. → pages 7, 44

[104] Keybase.io Home Page. https://keybase.io. Accessed: May 2017. → pages 53, 110

[105] Kiwi IRC Home Page. https://kiwiirc.com. Used version 0.9.0. Accessed: July 2017. → page 48

[106] How can I control how much data Netflix uses? https://help.netflix.com/en/node/87. Accessed July 2017. → page 106

[107] niltalk Terms & Privacy Policy. https://niltalk.com/pages/privacy. Accessed: July 2017. → pages 7, 44

[108] noteshred: Home Page. https://www.noteshred.com/client-side-encryption. Accessed: July 2017. → pages 7, 44

[109] Did the FBI Pay a University to Attack Tor Users? https://blog.torproject.org/blog/did-fbi-pay-university-attack-tor-users, November 2015. Accessed: July 2017. → page 80

[110] OpenPhoto Project Home Page). https://web.archive.org/web/20131112010343/http://theopenphotoproject.org/. Accessed: July 2017. The OpenPhoto project was rebranded as Trovebox in January 2013, and shut down in March 2015. → page 40

[111] Pew Internet Report: Americans and text messaging. http://www.pewinternet.org/files/old-media//Files/Reports/2011/Americans%20and%20Text%20Messaging.pdf, September 2011. Accessed: July 2017. → page 102

[112] S. Musil. Instagram competitor PicPlz to shut down in July. http://news.cnet.com/8301-1023_3-57446282-93/instagram-competitor-picplz-to-shut-down-in-july/, June 2012. Accessed: July 2017. → pages 11, 12

[113] Priv.ly project Home Page. https://priv.ly. Accessed: July 2017. → pages 49, 74

[114] WhatsApp Privacy Policy. https://www.whatsapp.com/legal/#privacy-policy, August 2016. Accessed: July 2017. → page 109

[115] Open Whisper Systems: Advanced cryptographic ratcheting description. https://whispersystems.org/blog/advanced-ratcheting/, November 2013. Accessed: July 2017. → page 109

[116] Open Whisper Systems: Technical Docs. https://whispersystems.org/docs/, 2017. Accessed: May 2017. → page 109

[117] Global Internet Phenomena Report - 1H 2014. https://www.sandvine.com/downloads/general/global-internet-phenomena/2014/1h-2014-global-internet-phenomena-report.pdf, May 2014. Accessed: July 2017. → page 108

[118] Global Internet Phenomena Report Latin America and North America - 2016. https://www.sandvine.com/downloads/general/global-internet-phenomena/2016/global-internet-phenomena-report-latin-america-and-north-america.pdf, June 2016. Accessed: July 2017. → page 108

[119] ShadowCrypt GitHub page. https://github.com/sunblaze-ucb/shadowcrypt. Used commit fa2de4e61dd7c0f18d177ba4c810dc935c68f32d. Accessed July 2017. → page 74

[120] Stanford Javscript Crypto Library (SJCL) Home Page. http://bitwiseshiftleft.github.io/sjcl/, . Beeswax uses version 1.0.0 compiled with AES and ECC. Accessed: July 2017. → page 68

[121] Stanford Javscript Crypto Library (SJCL) Home Page. http://bitwiseshiftleft.github.io/sjcl/, . DistoRt uses version 1.0.6 compiled with ECC. Accessed: July 2017. → page 104

[122] Slack Privacy Policy. https://slack.com/privacy-policy. Accessed: July 2017. → pages 7, 44

[123] Subrosa.io GitHub Page. https://github.com/subrosa-io/. Accessed: July 2017. → pages 7, 44

[124] Google. Launching Google Takeout. https://web.archive.org/web/20111021054234/http://www.google.com:80/goodtoknow/manage-data/takeout/, October 2011. Accessed: July 2017. Google users can now login to https://google.com/takeout. → page 13

[125] Twitter: API Rate Limits. https://dev.twitter.com/rest/public/rate-limiting, June 2016. Accessed: July 2017. → page 107

[126] Public Search API for tweets (developer docs). https://dev.twitter.com/rest/reference/get/search/tweets, February 2015. Accessed: July 2017. → page 59

[127] Twitter Usage Statistics. http://www.internetlivestats.com/twitter-statistics/, June 2016. Data from June 30, 2016. → pages 94, 107

[128] Unhosted: Freedom from web 2.0's monopoly platforms. http://unhosted.org. Accessed: July 2017. → page 39

[129] W3C. Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/, January 2014. Accessed: July 2017. → page 18

[130] W3C. Content Security Policy Level 2 (W3C recommendation). http://www.w3.org/TR/2015/CR-CSP2-20150219/, February 2015. Accessed: May 2015. → page 76

[131] W3C. DOM Level 3 Event Flow (UI Events). (working draft). https://www.w3.org/TR/2015/WD-uievents-20151215/#event-flow, December 2015. Accessed: July 2017. → page 61

[132] W3C. HTML5 Offline Web Applications (working draft). http://www.w3.org/TR/2011/WD-html5-20110525/offline.html, November 2011. Accessed: July 2017. → pages 30, 32

[133] W3C. Shadow DOM (working draft). https://web.archive.org/web/20170609075055/http://www.w3.org/TR/2014/WD-shadow-dom-20140617/, June 2014. Accessed: July 2017. → page 51

[134] W3C. Web Cryptography API, Candidate Recommendation. https://www.w3.org/TR/2014/CR-WebCryptoAPI-20141211/, December 2014. Accessed: July 2017. → page 68

[135] L. Tolstoy. War and Peace (on Project Gutemberg). http://www.gutenberg.org/ebooks/2600, 1869. → page 102

[136] Mozilla. Web Application Manifest Format and Management APIs (W3C editor's draft). http://mozilla.github.com/webapps-spec/, September 2012. Accessed: July 2017. Recent versions of working draft available at https://www.w3.org/TR/appmanifest/. → page 19