Towards A Non-tracking Web

Thesis approved by the
Department of Computer Science
of the University of Kaiserslautern (TU Kaiserslautern)
for the award of the Doctoral Degree
Doctor of Engineering (Dr.-Ing.)

by
Mr İstemi Ekin Akkuş

Date of the oral defense: 21.03.2016
Dean of the department: Prof. Dr. rer. nat. Klaus Schneider
1. Reviewer: Prof. Dr. Paul Francis
2. Reviewer: Dr. Deepak Garg
3. Reviewer: Dr. Claude Castelluccia

D 386

Towards A Non-tracking Web

by

İstemi Ekin Akkuş

To Emilia, my family and friends...

Acknowledgements

First and foremost, I would like to thank my advisor, Paul Francis. He allowed me to explore and find my own path through the unknown. When I got lost, his feedback and comments made me find the next steps. Every time I talked to him, I felt that I learned something new, obtained a new perspective and came to understand something that I did not understand before. I am forever grateful.

I also want to thank Deepak Garg. He was more than willing to spend time and effort to help me learn new tools and techniques. He provided invaluable feedback on my work and how I could improve it.

Rose Hoberman taught me a huge amount on how to communicate my ideas and thoughts. She helped me learn how to write papers and present them. All suggestions she made not only improved my work, but also helped me to think in different ways and organize my thoughts.

Saikat Guha, my host at Microsoft Research India, has helped me explore interesting ideas. During my internship, I learned not only about new technical tools, but also on different approaches to solve problems. I would like to also thank my host at the International Computer Science Institute, Nicholas Weaver, for enabling me to pursue and implement my ideas that later became part of this thesis.

All MPI-SWS faculty, staff, students and postdocs have made my time there enjoyable, be it through academic discussions, social events or just brief interactions. I thank to all those with whom my time at MPI-SWS has coincided. Special thanks to my office mate, Alexey Reznichenko, with whom I had a considerable amount of discussions about research and life in general. Similar discussions were also held with Scott Kilpatrick, Pedro Fonseca, Beta Ziliani, Georg Neis, Nuno Santos, Cheng Li, Manohar Vanga, Eslam Elnikety, Pramod Bhatotia, Bimal Viswanath, Anjo Vahldiek-Oberwagner, Reinhard Munz, Lisette Espín, Nancy Estrada, Juhi Kulshrestha, Alexander Wieder, Mainack Mondal, Ezgi Çiçek, Paarijaat Aditya, Arpan Gujarati, Aastha Mehta, Felipe Cerqueira, Viktor Erdélyi, Johannes Kloos, Natacha Crooks, Ruichuan Chen, Stevens Le Blond, Allen Clement, Arthur Charguéraud, Matthew Hammer and Roly Perera. All our interactions made me feel that I am not alone in this journey, and for that, I thank them.

I thank Vera Laubscher, Susanne Girard, Maria-Louise Albrecht, Corinna Kopke, Claudia Richter, Brigitta Hansen and Roslyn Stricker from the office staff as well as Christian Mickler, Tobias Kaufmann, Carina Schmitt and Christian Klein from the IST staff. While my requests never seemed to stop, they always found ways to accommodate me. More than that, I enjoyed our chats and interactions.

My family has always supported me no matter what (and I know they will continue to do so): my mom, dad, sister as well as my uncles, aunts and cousins. Their love made me feel safe. Their encouragements gave me strength. Every time I talked with them, I instantly felt better. I only wish that my grandma would have also seen this and that she would have been smiling.

I would like to specially thank my friends, Hüseyin and Zeynep Çankaya, for being there for me when I needed a break and for our trips to various locations to experience the world. I only hope that we will have more of that in the future. I also would like to thank Alex Kantchelian for interesting discussions and outings when I was in Berkeley. It certainly made my stay much more fun and enjoyable.

Last but the most, I would like to express my endless gratitude for Emilia Esposito. Without her love, support and care, this thesis would never have been completed. She made me smile when I was crying. She held my hand when I was afraid. She gave me a hug when I needed it the most. I can only hope to be as kind as she is. Her existence and smile were the biggest motivations for me to make progress.

Abstract

Today, many publishers (e.g., websites, mobile application developers) commonly use third-party analytics services and social widgets. Unfortunately, this scheme allows these third parties to track individual users across the web, creating privacy concerns and leading to reactions to prevent tracking via blocking, legislation and standards. While improving user privacy, these efforts do not consider the functionality third-party tracking enables publishers to use: to obtain aggregate statistics about their users and increase their exposure to other users via online social networks. Simply preventing third-party tracking without replacing the functionality it provides cannot be a viable solution; leaving publishers without essential services will hurt the sustainability of the entire ecosystem.

In this thesis, we present alternative approaches to bridge this gap between privacy for users and functionality for publishers and other entities. We first propose a general and interaction-based third-party cookie policy that prevents third-party tracking via cookies, yet enables social networking features for users when wanted, and does not interfere with non-tracking services for analytics and advertisements. We then present a system that enables publishers to obtain rich web analytics information (e.g., user demographics, other sites visited) without tracking the users across the web. While this system requires no new organizational players and is practical to deploy, it necessitates the publishers to pre-define answer values for the queries, which may not be feasible for many analytics scenarios (e.g., search phrases used, free-text photo labels). Our second system complements the first system by enabling publishers to discover previously unknown string values to be used as potential answers in a privacy-preserving fashion and with low computation overhead for clients as well as servers. These systems suggest that it is possible to provide non-tracking services with (at least) the same functionality as today's tracking services.

Contents

List of Figures **xiii**

List of Tables **xiv**

1 Introduction **1**
1.1 An Evolving Web 1
1.2 Essential Web Services & Third-party Tracking 2
1.3 Thesis Research & Contributions: Non-tracking Web Systems 5
1.4 Organization 8

2 A General and Interaction-based Third-party Cookie Policy **10**
2.1 Introduction 11
2.2 Contributions 13
2.3 Goals 14
2.4 Assumptions 14
2.5 Design 16
2.5.1 Detecting Third-party Requests & Removing Cookies 16
2.5.2 Click as User Interaction and Reload-on-click 17
2.5.3 Two-click Control for Social Widgets 17
2.5.4 Generalization 19
2.5.5 Social Widgets versus Advertisements 19
2.5.6 Third-party Cookie Access 21
2.5.7 Limitations 21
2.6 Implementation 22
2.7 Evaluation 24
2.7.1 Methodology 24
2.7.2 Efficacy of the Heuristic 24
2.7.3 Performance Overhead 33
2.8 Discussion 37
2.8.1 Robustness of the Heuristic 37
2.8.2 Reloading Advertisement Iframes with Cookies 38
2.8.3 Evercookies 39
2.8.4 Cookie Synching 39
2.8.5 Behavioral Advertisements & Extended Web Analytics 39

2.9 Conclusion & Future Work 40

3 Non-tracking Web Analytics **42**
3.1 Introduction 43
3.2 Contributions 45
3.3 Definitions & Components 46
3.4 Goals 47
3.4.1 Functionality Goals 47
3.4.2 Privacy Goals 48
3.5 Assumptions 49
3.5.1 Client 49
3.5.2 Data Aggregator 50
3.5.3 Publisher 51
3.5.4 Incentives 52
3.6 System Overview 52
3.6.1 Audits 55
3.7 Design 56
3.7.1 Differential Privacy Background 57
3.7.2 Queries 58
3.7.3 Query Response 60
3.7.4 Audit Response 61
3.7.5 Noise Generation 64
3.8 Analysis 68
3.8.1 Data Aggregator 68
3.8.2 Publisher 69
3.8.3 Client 75
3.9 Implementation & Evaluation 75
3.9.1 Implementation 75
3.9.2 Example Scenario 76
3.9.3 Deployment 79
3.10 Conclusion & Future Work 80

4 Privacy-preserving String Discovery **83**
4.1 Introduction 84
4.1.1 Background: Privacy-preserving Analytics Systems 85
4.2 Contributions 87
4.3 Definitions & Components 90
4.4 Goals 91
4.4.1 Privacy Goals 91
4.4.2 Functionality Goals 93
4.5 Assumptions 93
4.5.1 SplitX 94
4.5.2 Client 94
4.5.3 Aggregator & Proxies 94

4.5.4 String Values . . . 95
4.6 System Overview . . . 95
4.7 Building Blocks . . . 98
4.7.1 XOR-Encryption: Split & Join . . . 98
4.7.2 Blind Comparison via pairwise-XOR and hash (PXH) . . . 100
4.7.3 Noisy Threshold . . . 101
4.8 Design . . . 101
4.8.1 Initializing String Discovery . . . 102
4.8.2 Collecting Encrypted Strings . . . 104
4.8.3 Blindly Comparing & Counting Strings . . . 105
4.8.4 Mirror Operation & Oblivious Noise . . . 108
4.8.5 Other Details . . . 110
4.9 Optimizations . . . 110
4.9.1 Sample-Identify-Count-Filter (SICF) . . . 110
4.9.2 Short Hashes . . . 111
4.10 Detecting Duplicates . . . 113
4.11 Analysis . . . 115
4.11.1 Tools . . . 116
4.11.2 Modeling Primitives . . . 119
4.11.3 Protocol Model & Verification Results . . . 127
4.11.4 Informal Analysis of the Remaining Protocol . . . 136
4.12 Evaluation . . . 139
4.12.1 Datasets . . . 140
4.12.2 Benefits of Optimizations . . . 141
4.12.3 Microbenchmarks . . . 142
4.12.4 Experiments with Real-world Data . . . 144
4.13 Conclusion & Future Work . . . 147

5 Literature Survey **148**
5.1 Third-party Tracking . . . 148
5.2 Web Analytics . . . 152
5.3 Privacy-preserving String Discovery . . . 155

6 Summary & Final Remarks **159**

Bibliography **163**

Curriculum Vitae **179**

List of Figures

1.1 Third-party tracking by data aggregators, online social networks and advertisement networks. 3

2.1 Priv3+ options in Mozilla Firefox. 23
2.2 Priv3+ options in Google Chrome. 23

3.1 Operation of today's tracking web analytics systems. 47
3.2 Query workflow of our system. 54
3.3 Auditing mechanism of our system. 56
3.4 Confidence level for the isolation attack via dropping target's answer. . . 72
3.5 Probability density function of Laplace noise values. 80

4.1 Overview of our system's operation. 96
4.2 Splitting and joining. 98
4.3 Mirror operation 1 in our privacy-preserving string discovery system. . 102
4.4 Initializing string discovery. 103
4.5 Collecting encrypted strings. 103
4.6 Counting and revealing strings. 104
4.7 Complete system for privacy-preserving string discovery. 109
4.8 Overview of our SICF heuristic. 112
4.9 An example of a constructor without a destructor: one-way hash function. 117
4.10 An example of a constructor with a destructor: symmetric encryption . . 117
4.11 An example of a constructor with equations: symmetric encryption . . . 118
4.12 Formal definition of splitting and joining. 120
4.13 Formal definition of the PXH operation. 121
4.14 Special channels to emulate the PXH comparison due to incomplete XOR-functionality support in ProVerif. 123
4.15 Use of the special channels in each process. 123
4.16 Formal definition of linkability. 125
4.17 Formal definition of existence. 126
4.18 Adversary queries in our ProVerif model. 129
4.19 Speedup vs. number of hash buckets. 142
4.20 CPU time vs. number of strings. 146

List of Tables

2.1 Crawls & datasets for our evaluation. 25
2.2 Number of iframes recorded in successfully loaded pages. 27
2.3 Distribution by domain and category of single, third-party iframes. . . . 28
2.4 Distribution by domain and category of nested, third-party iframes. . . . 32
2.5 Number of nested iframes belonging to an online social network widget. 33
2.6 Distribution of iframes from social widget domains. 34
2.7 Distribution of iframes from advertisement domains. 35
2.8 Number of successfully loaded links. 36
2.9 Performance overhead of cookie policies. 37

3.1 Query fields 58
3.2 Queries and associated parameters. 77
3.3 Bandwidth usage of the publisher and the data aggregator. 77
3.4 Number of clients having used search engines during our deployment. . 78

4.1 Process grammar in applied pi calculus. 116
4.2 Attacks found by ProVerif by various adversaries. 128
4.3 Client microbenchmarks. OT assumes 32-bit strings. Other operations assume 100-byte strings. 143
4.4 Server microbenchmarks. OT assumes 32-bit strings. Other operations assume 100-byte strings. 144
4.5 Memory overheads for privacy-preserving string discovery. 144

Chapter 1

Introduction

1.1 An Evolving Web

Ever since the beginning of the web, statistics about visitors or *web analytics* have been important. Website publishers use web analytics information to analyze their traffic and optimize their sites' content accordingly. To achieve this goal, numerous analytics software programs have been developed and have been at the disposal of publishers since the early days [7,28,40,41,67]. By running these programs, publishers can obtain statistics about users on their site, such as their page views, clickstreams, browsers, operating systems and plugins as well as the visit frequency of returning visitors.

In recent years, the prevalence of mobile devices and the number of users accessing the web using these devices have increased drastically. While the traditional web analytics software can still be used for website visits with a browser running on a mobile device, such devices are often used to access content and services of publishers with stand-alone applications. As a result, the number of services that target specifically these mobile applications and the analytics information has dramatically increased. Some examples of these services are Flurry, Bango Dashboard, Localytics, Countly, Mobclix and Google mobile analytics [8,13,21,35,37,38].[1]

Another important aspect the publishers have almost always been interested in is the interaction with their visitors. From simple guestbooks, where visitors could leave a short message for the publishers, to forums and comments, where visitors could interact with the publishers as well as other visitors, such interactive features have always been an integral part of websites. Using the analytics tools described above, publishers can track popular pages on their own sites via user interaction (e.g., comments, page visits), and use this information to suggest new and interesting content. Perhaps, the continuation of these features is the increasing prevalence of social widgets present on web pages today [69,144], which not only enable publishers and visitors to interact with each other, but also allow visitors to share their interaction with other users.

1.2 Essential Web Services & Third-party Tracking

Today, however, publishers often outsource the above services to a third-party service provider. This outsourcing is convenient for publishers, because they only have to install a small piece of code (i.e., a JavaScript code snippet) provided by the service provider. More importantly, this arrangement fills a gap in the existing first-party approaches for web analytics or social interactions with users: First-party analytics solutions do not provide potentially useful information, such as demographics of visitors (e.g., age, gender, education level) or other sites visited by them. Similarly, interaction between publishers and visitors through comments and page visits is usually limited to existing visitors, and does not cover potential new users.

To obtain *extended web analytics* (e.g., user demographics, other sites visited), web

[1]There is no fundamental reason why software developers for regular desktop computers cannot use such data aggregators to obtain analytics information.

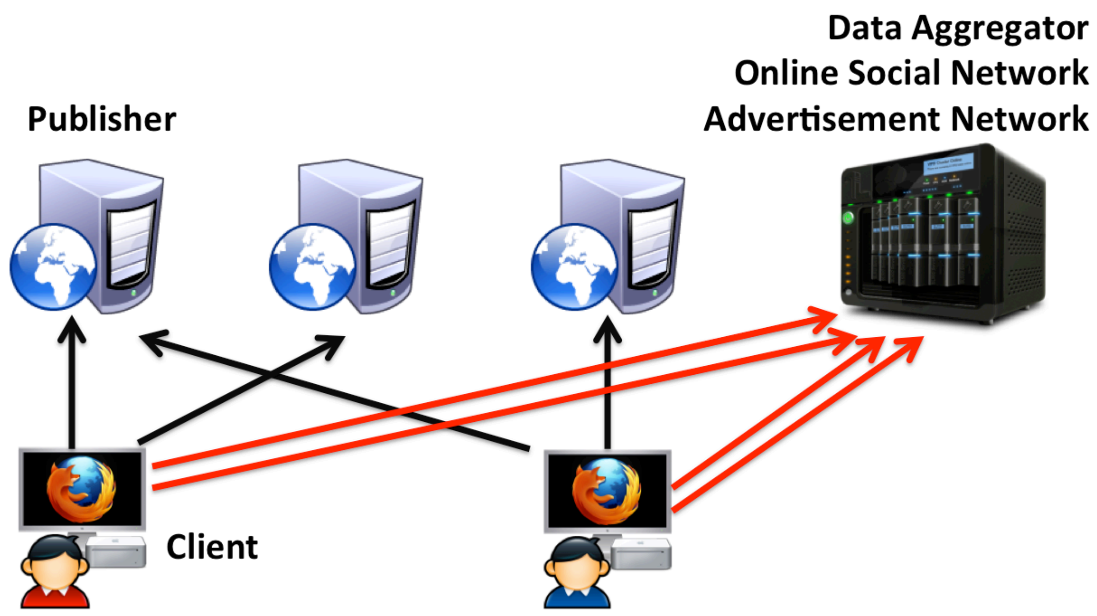


Figure 1.1: Third-party tracking by data aggregators, online social networks and advertisement networks.

publishers often use data aggregators such as comScore, Google or Quantcast [6, 62]. Such a data aggregator collects data from visitors of many publishers. This information enables a data aggregator to *infer* extended web analytics information that goes beyond traditional web analytics and includes user demographics. The aggregator then provides the publishers with the analytics information of their sites in aggregate form. These aggregators may also provide behavioral advertisements that are tailored to a user's interests based on her browsing history.

For interactions with visitors, publishers often use online social network (OSN) services and employ social widgets from OSN providers, such as Facebook and Twitter. These widgets help publishers to increase user engagement on their sites. Additionally, these widgets increase the exposure of the site to more potential visitors, because the interaction with social widgets propagate this action to more users via the OSN's structure.

This arrangement benefits not only the website publishers, application developers and other analysts, but also the data aggregators and OSN providers: Every time a user visits a publisher website, the data aggregator or the OSN provider learns about this visit due to the loading of the third-party resource embedded on the publisher website (e.g., a JavaScript code snippet, social widget) and can track the user across the web (Figure 1.1). As a result, these entities obtain vast amounts of information about users' browsing behavior across the web and use this information via targeted advertisements [16]. Compiling extended web analytics benefits the data aggregators because they can sell this information to advertisers and publishers alike. Learning the interactions with different publishers, the OSN providers get more detailed information about a user's behavior, which then can be combined with the information the user has provided while using the OSN.

This arrangement, however, comes with a price for user privacy and raises concerns about users being tracked while accessing publisher content. Such tracking enables these third-party service providers to compile detailed behavior of individual users with the sensitive information they obtain, and infer individual–not just aggregate–user demographics [144]. Thus, these third parties are given a lot of information about users' actions on the web and have to be trusted that they will not abuse it. This trust has been violated in the past [33,47,49].

Tracking affects not only users, but also the data aggregators and OSN providers themselves, who are often criticized for privacy violations due to tracking. In response to these criticisms, researchers and industry have proposed methods to detect and prevent tracking, including voluntary regulations by the industry to provide opt-out mechanisms [10,11,50,66], the Do-Not-Track (DNT) initiative in the W3C and US FTC [55], and many client-side tools, either to implement DNT [24,52,57], or to outright prevent tracking using blacklists, which also include social widgets [15,22,23,29,55,68,70,144].

While these methods protect privacy, they significantly reduce the benefits of analytics and social widgets by limiting the information publishers can learn about their visitors and by limiting the interaction between the visitors and the publisher, respectively. To the extent that these efforts take hold, the ability for data aggregators to provide extended analytics to publishers will be degraded, and OSN providers will no longer be able to provide an appealing service for publishers to interact with their visitors and attract new users.

1.3 Thesis Research & Contributions: Non-tracking Web Systems

In today's systems, user privacy suffers while publishers, aggregators and OSN providers mostly benefit from third-party tracking. In a web, where third-party tracking is prevented via the use of client-side tools, publishers' ability to learn statistics about their users to provide them with better and more content, to monetize that content via advertisements and to increase the exposure of their sites to more users will be hampered. We think that users and publishers both are crucial parts of the current web, and failure to satisfy one side's needs will lead to an unsustainable web ecosystem.

In this thesis, we take the stance that just preventing third-party tracking of users without considering the functionality it serves to web publishers and other entities is not a viable solution. To this end, our research is concerned with the design, analysis and implementation of practical systems with the goal of preserving the functionality of today's systems as much as possible, but do so without tracking users. More specifically, we want to provide a method for enabling social interactions on websites in an on-demand fashion and a system that provides extended web analytics in aggregate form to publishers and data aggregators without violating individual user privacy. Overall, this thesis makes the following contributions:

A General and Interaction-based Third-Party Cookie Policy: We present a new third-party cookie policy for modern browsers and explore its implications. In this new policy, third-party cookies are only sent to the third-party domain when (and if) the user interacts with the third-party content.

We consider this interaction as the user's intent to engage with the third-party content, and thus, willingness to let the third-party know about the current page the user is visiting (i.e., on the first-party domain). This approach prevents undesired third-party tracking with cookies, but enables interactive features such as social widgets on-demand. As a result, the user is given more control. While other approaches such as anti-tracking browser add-ons and other tools are similar in this regard, our policy is general and does not require a curated blacklist that can be difficult to maintain.

Furthermore, the blacklisting tools above often consider non-tracking services for analytics and advertisement services harmful and block them. As a result, publishers who opt to be more privacy-friendly in their practices get penalized: their ability to obtain statistics about their users and to monetize their content through advertisements is hurt. In comparison, our policy does not interfere with non-tracking services for analytics and advertisements.

Non-tracking Web Analytics: We describe the design, analysis and implementation of the first web analytics system that can provide publishers with better extended web analytics (e.g., user demographics) while eliminating the need for third-party tracking and providing users with privacy protections (e.g., anonymity, unlinkability). While doing so, our system does not introduce a new organizational component into the ecosystem, which makes it practical and easier to adopt.

The approach we are taking is to store user data on the user device under the user's control with the help of a client software (e.g., browser). This data is queried directly by the publishers, who distribute queries to the browsers when the users visit the publisher website. After the clients execute the queries, they encrypt the answers with the data aggregator's public key. The publishers then relay the encrypted answers to the data aggregator, who decrypts them, compiles the final result and shares it with the publisher. During this process, both entities add differentially-private noise.

Keeping the user data on the user device enables us to get a more complete picture of user's browsing activities as well as to have more accurate demographic information. Combined with the direct measurement of this information and the addition of differentially-private noise, this approach enables our system to provide more accurate and more types of extended web analytics than today's systems without violating individual user privacy.

Privacy-preserving String Discovery for Non-tracking Analytics: While the above system is effective at eliminating the need for third-party tracking for analytics purposes, it achieves this goal with one caveat: the system depends on the existence of pre-defined string values as potential answers for queries distributed by the publishers. As a result, publishers would have to enumerate a list of such values depending on the queries they would like to distribute.

This requirement may not be an issue for certain types of queries, in which the potential answer values are of small types, such as gender, education level and age. On the other hand, queries about websites visited, search phrases used or free-text tags assigned to photos may become a problem. In these cases, the anticipation and enumeration of potential answer values may be difficult or impossible, limiting the applicability and effectiveness of the analytics system.

We describe the design, analysis and feasibility study of a privacy-preserving string discovery system that enables analysts to discover previously unknown string values.

Our system is complementary to the above non-tracking analytics system as well as other similar systems [97,98,117,128].

Similar to the above systems, the user strings are stored on the user device with a client software. During the discovery, the clients send their encrypted strings for aggregation by an aggregator and two proxies. The encrypted strings are then counted blindly, such that the string values are not revealed during the counting. Afterwards, noise is added to the counts. String values whose noisy counts pass a discovery threshold are then revealed.

The user devices running the client software may have limited power and bandwidth resources (e.g., mobile devices). To support even such clients, we employ low-cost primitives (e.g., XOR-encryption). Our novel blind comparison method enables our system to count encrypted strings without decrypting them. Besides the reduced client overhead, these low-cost primitives also reduce the computation overhead on the server components, with the caveat of increased but still cheap bandwidth overhead. In other words, our system trades off cheap server bandwidth for drastically reduced client and server computation overhead.

1.4 Organization

The rest of this thesis is as follows. In Chapter 2, we propose, prototype and evaluate a new, general and interaction-based cookie policy that prevents third-party tracking without penalizing common web services such as social networking. In Chapter 3, we present the design and analysis of a novel system that eliminates the need for third-party tracking on the web and provides publisher websites as well as data aggregators with better extended web analytics. In Chapter 4, we describe the design, analysis and evaluation of a privacy-preserving string discovery system that complements ours (and similar) non-tracking analytics systems by providing them with potential answer values for their queries. We discuss related work in Chapter 5 and conclude in Chapter 6.

Chapter 2

A General and Interaction-based Third-party Cookie Policy

In this chapter, we propose and explore a new and general third-party cookie policy that considers user interaction. This policy gives users more control regarding tracking, does not require a curated and maintenance-requiring blacklist, and does not hamper functionality of essential web services. A preliminary version of this work without the evaluation was published in the IEEE Workshop on Web 2.0 Security and Privacy (W2SP) in 2015 [82].

This chapter is organized as follows. The next section motivates the need for a new and general cookie policy that considers user interaction when dealing with third-party content. Section 2.2 presents our main contributions in devising such a policy. We list our goals and our assumptions in Sections 2.3 and 2.4, respectively. Section 2.5 explains our design and how it achieves our goals. We describe our implementation in Section 2.6. Section 2.7 presents the evaluation of our implementation's functionality and efficiency using several thousands of pages from top 10K most popular sites from Quantcast. We discuss our policy's implications in Section 2.8 and conclude in Section 2.9.

2.1 Introduction

Tracking on the web by third-party service providers such as data aggregators and online social network (OSN) providers raises concerns about user privacy. Existing cookie policies in modern browsers offer a partial solution to this problem. A user can select a cookie policy, in which the browser will not allow any third-party cookies to be set. While effective at preventing tracking for behavioral advertisements, this policy causes issues when the third party is an OSN provider. For example, Mozilla Firefox's "never accept third party cookies" policy breaks the functionality of social widgets on publisher sites and does not allow a user to interact with them, even when the user is logged in to the OSN that provided the widgets. Google Chrome and Apple Safari behave the same way.

Another privacy option in Firefox aims to solve exactly this problem, such that cookies from third parties will be accepted if the user has visited the third party site as a first party in the past.[1] For example, if the user is logged in to Facebook, all Facebook Like buttons on other publishers will function properly. This option, however, allows the OSN provider to act as a tracker and learn about the user's visit to the publisher, even if the user did not interact with the social widget [100, 144]. Tools, such as Priv3 [43, 100] and ShareMeNot [53, 144], aim to prevent this tracking by OSN providers.

Following these tools, other popular client-side privacy tools like Ghostery [23] and Disconnect [14] started preventing social widgets from being loaded, in addition to the trackers by data aggregators. Using a blacklist of aggregators, behavioral advertisers and OSN providers, these tools scan the loading page and prevent blacklisted elements

[1]Similar to Safari's "block cookies from third parties and advertisers".

from being loaded.

This blacklist-based approach has several limitations. First of all, the blacklist needs to be maintained and then distributed to clients in a timely fashion; otherwise, the benefits of using such a tool are greatly reduced. These tools try to find an optimum update schedule for their tracker libraries. For example, Ghostery regularly (e.g., every few weeks) updates its library of trackers while Disconnect checks for updates every day. Other less popular tools like Priv3 and ShareMeNot support only a handful of third parties, and depend on their developers to keep up with new social widgets. This maintenance of the blacklist can be cumbersome and error-prone: there is no guarantee that *all* third-party trackers will *always* be included in the blacklist.

In addition, these blacklists are very broad: They include first-party analytics tools, such as Piwik, Open Web Analytics and Mint Analytics [36,40,41], preventing publishers from learning about their visitors' behavior on their own sites. These blacklists can and do also include non-behavioral advertising that, by definition, does not require the tracking of users across the web (e.g., Project Wonderful). As a result, publishers who choose more privacy-friendly solutions for analytics and advertisements by *not* using third-party tracking are being unnecessarily penalized. While such non-tracking services can be removed from the blacklist, the maintenance issue is only amplified: the tool provider now has to categorize and determine which solutions are acceptable.

Finally, such a blacklist may be bypassed by third parties, simply with a configuration trick at their servers. For example, Apache2 directive 'AliasMatch' [39] enables a third party to serve the blacklisted element (i.e., JavaScript file, social widget) via customizable URLs, such that each publisher uses a different source, yet the third party serves the same file. This trick would force the tools to blacklist entire domains, which can become problematic if legitimate files not related to tracking are also served (e.g., libraries, images, OSN site).

2.2 Contributions

In this chapter, we propose and explore a new third-party cookie policy. The main idea is to send the associated cookies of third-party domains in a page only after the user has interacted with the third-party content.

Imagine the user is visiting a page on *siteA* (the first-party domain), which embeds a resource from *siteT* (the third-party domain). *siteT* then sets a cookie value to the user's browser to indicate that the user visited *siteA*. Later, when the user visits another page on *siteB* that also embeds a resource from *siteT*, *siteT* would normally receive the cookies it set before, and thus, learn that this user has visited both sites, *siteA* and *siteB*.

With our policy, the browser would not send any cookies associated with *siteT* while loading the third-party resource on *siteB*. While this idea is simple, it is effective to prevent third-party tracking with cookies: if *siteT* does not receive its cookies during the second load of the resource, it will not know that this user has been to *siteA*.

Simply preventing third-party tracking, however, is not a viable option. In today's web, the resource in the above example might be a social widget that allows the user to share web content with her friends. Our goal is to design a general solution that will not only prevent third-party tracking, but do so without penalizing essential web services such as social networking, advertisements and analytics.

To support such cases, we augment our policy such that these third-party resources are reloaded after user interaction, but this time while sending the associated third-party cookies. As a result, the functionality of social widgets is preserved both for publishers and users, and enabled on-demand for users.

While the reload-upon-interaction idea is not new and have been used in multiple client-side tools, all these tools require a blacklist. In comparison, our policy is general and is applied to *any* third-party content. This generalization coupled with user-specified whitelists gives users more flexibility as well as more control regarding the amount of tracking they would want and when it can occur.

2.3 Goals

We advocate that the users should be the final decision makers with regards to tracking by third parties, be they data aggregators or OSNs. Specifically, we would like our policy to enable interactive features (i.e., social widgets) in an on-demand fashion. Like previous approaches [43, 100, 144], we think that user interaction is necessary to achieve balance for privacy and functionality for a social web.

At the same time, we would like to devise a general cookie policy to prevent third-party tracking by not only OSNs but also data aggregators and advertisers. This policy should not depend on a blacklist unlike the above tools; thus, it should not require the cooperation of a developer to maintain and distribute such a blacklist to protect user privacy.

Finally, our policy should not interfere with non-tracking services for analytics and advertisements, and penalize publishers using such arguably more privacy-friendly services. We recognize the fact that for a sustainable web, publishers need statistical data about their users to improve their services as well as advertisements to financially support their operations.

2.4 Assumptions

When a user visits a web page, we consider the domain serving the page as the first party. All other domains are considered third-party domains. Although it is possible that some third-party domains belong to the same entity that owns the first party domain, we do not consider these cases. A method to dynamically determine if two domains belong to the same entity is orthogonal to our policy and is outside our scope.

We consider any content that is not loaded from the first party domain as third party content. Such content can include social widgets from OSN providers as well as advertisements.

We assume that the cookie preferences reflect the users' intentions and that the third parties are not going to try to bypass them. In a recent example, Doubleclick was caught deliberately circumventing Safari's default policy, and got sued by the Federal Trade Commission [25]. We assume such attempts are frowned upon, if not illegal, and the attempting party risks its reputation. We think that the data aggregators providing voluntary opt-out mechanisms already show their good faith in this regard, and that this assumption is reasonable.

More specifically, we leave methods to circumvent user cookie preferences outside our scope. One such method is fingerprinting, in which a third party creates a unique signature of a user's browser by combining various pieces of information in the browser environment, such as plugins, fonts and resolution. This fingerprint is then used to track the user across websites [106,131] without storing any cookies on the user's device. With the prevalence of such practices increasing [76,77,137], potential defenses are already being researched [111,136].

Another method outside our scope is 'cookie synching' [12,56]. In cookie synching, publishers share first-party cookie values of their users with third parties, by embedding a resource request to the third party with the first-party cookie values as parameters, enabling the third parties to set their own cookies. As a result, they can establish a mapping between the received cookie values and the cookie values they set, such that the user's browsing behavior can be correlated.

Finally, we assume that mashups, sites with data and code from multiple publishers, are interactive. If not, we assume that they will continue to function without the third parties receiving user-specific data (i.e., cookies). For example, most mashups using the Google Maps API still function if the user is not logged in to Google. If the mashup is interactive, our policy will reload the third-party content with associated cookies. As a result, the mashup should continue to function as expected.

2.5 Design

To achieve our goals, we propose the following policy: Any content from a third party domain (e.g., social widgets, advertisements) should be loaded without sending the associated (third-party) cookies. This content will be reloaded with the associated cookies, when (and if) the user interacts with it.

2.5.1 Detecting Third-party Requests & Removing Cookies

When a page is being loaded, the policy will check the HTTP GET requests for the resources embedded in the page. These resources can include images, scripts, embedded videos and iframes as well as resources embedded in iframes. Requests to the first-party domain are let through unchanged with their cookies. Requests for a resource from a third-party domain are only let through after removing the cookie values. These cookies are still present in the browser (i.e., they are not deleted), and used when the user interacts with the third-party content.

2.5.2 Click as User Interaction and Reload-on-click

We define user interaction as the mouse click to a page element. A click covers a big portion of user interactions with content, such as following a link, clicking an advertisement or bringing an element into focus. Previous tools also use a click as an indication of user intent to interact with social widgets [23,43,53].

Although other events such as key presses or hovering over an element can also constitute user interaction, such events can be more complex than a user click. For example, in the presence of multiple page elements that register key press event listeners, it is not exactly clear how to determine the interacted element without requiring extra effort by the user (e.g., a click). Similarly, hovering might not be easily distinguished from movement among page elements or the user being idle. On the other hand, a user click clearly defines the interacted element. We leave covering these other cases to future work.

Previous work, Priv3 [43,100], showed that reloading social widgets after the user click is effective for enabling social features on a website without compromising user privacy for functionality. For example, when the user wants to click the Facebook Like button on a page, it is reloaded by sending the user's Facebook cookies. In our design, we inherit this selective reload functionality. However, our work enhances this approach with two new mechanisms, which are described next.

2.5.3 Two-click Control for Social Widgets

After the third-party content is initially loaded without sending the user's cookies, we use a two-click control. The first click enables the third-party content by reloading it with the user's cookies. The second click registers the original action. For example, when the Facebook Like button is reloaded after the (first) user click, it shows information about the friends of the user who also liked that page. If the user wants to like the page, the second click will register the action. In this case, Facebook knows about the user's visit to that page only after the first click (i.e., activation of the widget) if the user chooses to click the widget. If not, it would not recognize this user as a logged-in user.

Enabling widgets in this manner still provides functionality, but at the same time, allows OSN users not interested in using the social widgets to have more control over when OSNs can learn about their browsing. Recently, the Belgian privacy commission took Facebook to court for the tracking of non-users as well as logged-out users by placing cookies not necessarily related to the login status of the user to Facebook [9, 17]. With our two-click control, such tracking by Facebook would not be possible.

It may be possible to combine the two clicks into one action, such that the activation of the widget by reloading it with cookies (i.e., first click) and the widget action (i.e., second click) can be triggered with a single user click.[2] However, we decided to use the two-click scheme, because it gives the user additional control regarding the functionality of the social widget: it is possible for the user to be interested only in the personalized content the OSN provides, without the user activating the social sharing feature. For example, the user may want to enable the 'Like' button to see which friends of her have liked a page (i.e., first click), but the user may not want to like the page (i.e., second click). With a single-click scheme, this scenario would not be possible.

The statistics about the number of loads of a widget compiled by the OSN provider may be inflated because of the reload. Strictly speaking, these statistics would be compiled using third-party tracking by the OSN provider: the statistics would not contain just the number of loads of a widget, but also the user as well as the page embedding the widget. Our policy is aimed to prevent this kind of behavior. Nevertheless, it is possible to augment our policy's reload functionality with extra information to indicate that the

[2]In fact, previous work, Priv3 [43, 100], uses this method.

new request is a reload (and not a new load) of the widget, such that OSN providers can correct their statistics about the number of loads. This augmentation does not give more information to the OSN provider about the user, because the reload already happens with user cookies after the user's interaction with the third-party widget.

2.5.4 Generalization

In contrast to previous approaches [14,23,43,53], our policy does not require a blacklist: it is applied to any third party content. Detecting such content is a straightforward task similar to the same origin policy already employed by browsers.

Besides social widgets, our policy is also effective in preventing other third party tracking via cookies: the user cannot interact with 'invisible' elements (e.g., pixel tags, invisible iframes) that are used for behavioral advertising and data aggregation purposes. As a result, cookies associated with these third parties will never be sent, preventing them from tracking the user across the web. These elements do not need to be detected at runtime or enumerated in advance as in a blacklist, because the policy applies to *any* third-party content.

Finally, our policy does not interfere with first-party analytics tools, because these tools use cookies that belong to the first-party domain whose requests are not modified. Similarly, this policy does not interfere with non-tracking (i.e., non-behavioral) advertisement systems, which by definition do not use any tracking cookies to load advertisements.

2.5.5 Social Widgets versus Advertisements

Third-party content such as social widgets and advertisements are loaded in a container, usually an iframe. The JavaScript in an iframe is prevented by the browser from accessing and manipulating the embedding page's document object model (DOM) tree. This model enables publishers to include content from third-party providers without having to worry about whether they are going to affect the embedding page's content.

The lack of a blacklist forced us to develop a heuristic to distinguish between advertisements and social widgets. A click for an advertisement needs to be passed unchanged, whereas a click for a social widget requires a reload. For this purpose, we make the following observation and validate it later in Section 2.7.

Social widgets are usually loaded in a single, non-nested iframe. For example, Facebook's Like button, Twitter's Tweet button and Google+'s +1 button as well as commenting platforms such as Disqus are usually loaded in a single iframe. The user click on this iframe simply triggers the reload of the iframe. Note that this approach is different from other client-side tools, such as Ghostery, which require the user to reload the entire page rather than a single element after the user interaction.

On the other hand, third-party advertisements are usually present in nested iframes: the advertisement is usually present in an iframe that is contained within another iframe. In these cases, the user click is just passed through without any changes and without reloading any of the iframes.

In our design, we use the following heuristic. We first check the target node of the user click and obtain the hyperlink. If there is no such node or no hyperlink is present, we consider the source of the iframe as the target. If the link belongs to a third party domain who had no cookies filtered with our policy, we let the click through. If the link belongs to a third party domain whose cookies our policy filtered, we check whether the target node is present in a single or nested iframe. If the target is in a single iframe, the iframe is reloaded with the corresponding cookies. If it is in a nested iframe, the click is passed through. As a result, clicks to social widgets should trigger a reload whereas clicks to advertisements should pass unchanged.

2.5.6 Third-party Cookie Access

Strictly speaking, our policy does not prevent third parties from setting cookies on the user's browser. One caveat of this approach is that they can receive these cookies later, if the user visits the third party site as a first party. This issue opens the possibility of the third party accumulating the browsing history of the user by setting its cookies and hoping the user visits its website. Although the probability of a user visiting a tracker's website may be low, this issue becomes more important if the third party is an OSN.

Previous work, Priv3 [43, 100], prevents third-party scripts from accessing cookie values until the user interaction. As a result, third parties cannot use JavaScript to compile a list of visited pages in the cookie values to receive them later. In our design, we also use this approach. This problem might also be solved by refusing new third-party cookies, but we leave it to future work.

2.5.7 Limitations

Our biggest limitation is that our heuristic may fail to distinguish a social widget and an advertisement loaded in a single (i.e., non-nested) iframe. A user click on the advertisement may trigger a reload, if the target node is a third party whose cookies our policy has filtered. This reload may have an adverse effect such as creating an extra impression that otherwise would not have occurred. More importantly, the click on the advertisement may not register creating an undesired behavior. We further investigate how prevalent this issue is in Section 2.7.

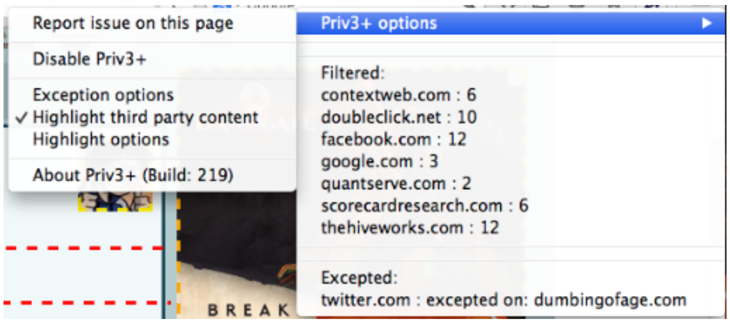
2.6 Implementation

We implemented a proof-of-concept of our proposed policy as browser extensions. Our implementation, Priv3+, available for Firefox [44] and Chrome [45]. To date, it has been downloaded about 28K times and has about 4K daily users.

Priv3+ inherits the selective reload functionality and the third-party cookie access mechanism from previous work, Priv3 [43,100], but implements the following additional features. First, Priv3+ generalizes the idea of removing third-party party cookies from all third-party resource requests and stores the corresponding page elements to reload them after user interaction. Second, it implements the two-click control, such that the first click on a widget first enables it and reloads third-party content with user cookies, and the second click registers the original action if the user chooses to do so.

Furthermore, Priv3+ shows information about the third-party domains and how many resources were loaded from each third party domain. It implements the whitelisting functionality and presents a basic graphical user interface to enable the user to add exceptions to the policy, such that certain third-party domains will be allowed to receive their cookie values on certain publisher websites without user interaction (Figure 2.1).

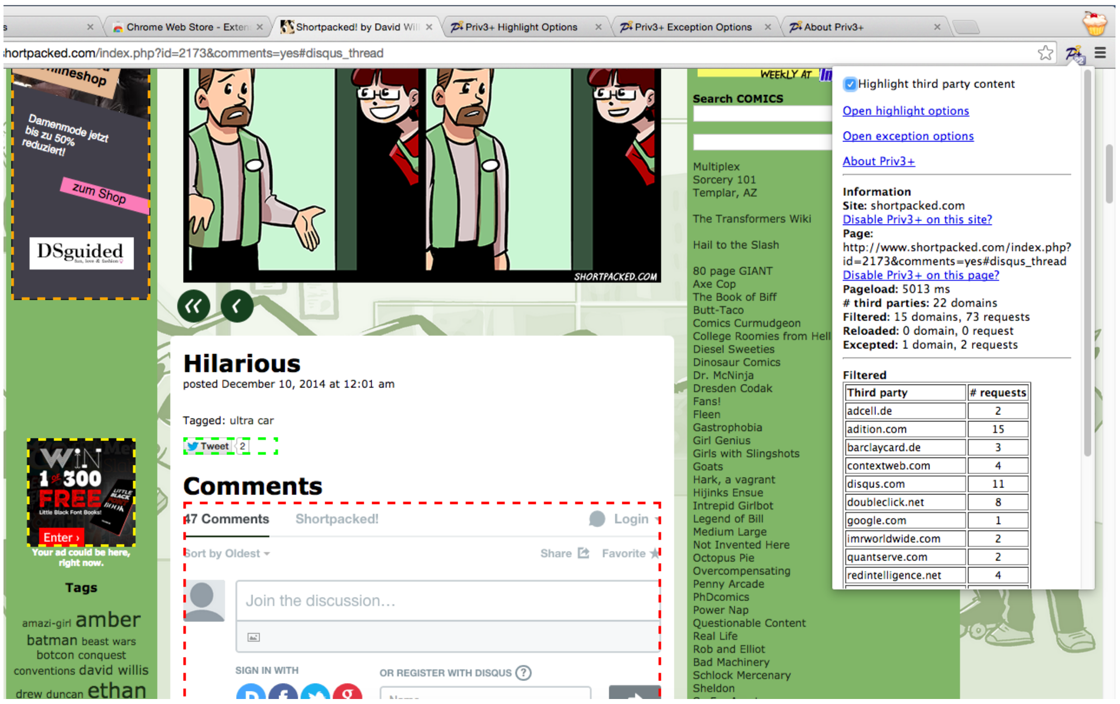
Priv3+ can highlight different types of third-party content. For example, Figure 2.2 shows dashed lines around the Tweet button, the comments by Disqus and the advertisements: Green indicates exceptions (i.e., the Tweet button). Red indicates third party resources in a single iframe (i.e., the Disqus comments). Orange indicates third party resources in nested iframes (i.e., the advertisement in the upper left). Yellow indicates a potential third party element whose source attribute was not present (i.e., the advertisement in the bottom left). Finally, Priv3+ shows a tooltip information about the request to load the third party content: whether the third party's cookies were removed during the request, or the third party did not have any cookies during the request.


Report issue on this page
Disable Priv3+
Exception options
✓ Highlight third party content
Highlight options
About Priv3+ (Build: 219)
Priv3+ options

```
Filtered:
contextweb.com : 6
doubleclick.net : 10
facebook.com : 12
google.com : 3
quantserve.com : 2
scorecardresearch.com : 6
thehiveworks.com : 12

Excepted:
twitter.com : excepted on: dumbingofage.com
```

Figure 2.1: Priv3+ options in Mozilla Firefox. Priv3+ shows information about the third party domains present on this page. The user can choose to add exceptions to the policy as well as highlight different types of third party content.



✓ Highlight third party content

Open highlight options

Open exception options

About Priv3+

Information
Site: shortpacked.com
Disable Priv3+ on this site?
Page: http://www.shortpacked.com/index.php?id=2173&comments=yes#disqus_thread
Disable Priv3+ on this page?
Pageload: 5013 ms
# third parties: 22 domains
Filtered: 15 domains, 73 requests
Reloaded: 0 domain, 0 request
Excepted: 1 domain, 2 requests

Filtered

Third party	# requests
adcell.de	2
adition.com	15
barclaycard.de	3
contextweb.com	4
disqus.com	11
doubleclick.net	8
google.com	1
imrworldwide.com	2
quantserve.com	2
redintelligence.net	4

Figure 2.2: Priv3+ options in Google Chrome. Priv3+ shows information about the third party domains present on this page. The user can choose to add exceptions to the policy as well as highlight different types of third party content. twitter.com is excepted by the user (i.e., green), the Disqus comments are present in a single iframe (i.e., red) and the advertisement in the upper left corner is present in nested iframes (i.e., orange). The advertisement in the bottom left is highlighted as a potential third party element (i.e., yellow) because the iframe did not have any source attribute.

2.7 Evaluation

In this section, we evaluate the effectiveness of our policy using large-scale web crawls. We first evaluate the effectiveness of our heuristic to distinguish social widgets and advertisements. We then report on the performance overhead of our policy.

2.7.1 Methodology

We created a new user profile in Firefox with the default cookie policy settings (i.e., "accept all cookies") and installed a simple Firefox add-on we created. This add-on automatically records the URL of the page and the page load time as well as the rendered page's DOM tree at the point when the JavaScript window.onLoad event is fired. Before we record the DOM tree, we wait an additional 3 seconds to allow some time for further resources to load, for instance, via asynchronous XMLHttpRequests.

For our crawls, we used popular websites from Quantcast [51]. We visited the sites and recorded the pages with our add-on. We allowed each site about 45 seconds to finish loading before moving to the next one. We then randomly extracted up to 10 links from each successfully loaded site. Table 2.1 summarizes our crawls and resulting datasets.

To simulate a user logged in to an OSN, we created user accounts on the following social network services: Google+, Facebook, Twitter, Disqus and Pinterest. Before we conducted a crawl, we manually logged in to these services and ensured that the cookie values persist (i.e., 'Remember me' option is selected).

2.7.2 Efficacy of the Heuristic

Our heuristic distinguishes social widgets from advertisements by checking whether the interacted element is in a single iframe. If so, the iframe is reloaded with appropri-

Table 2.1: Crawls & datasets for our evaluation.

Crawl name	Crawl 1 (prelim.)	Crawl 2	Crawl 3
Quantcast snapshot	Nov 2014	Mar 2015	Mar 2015
Dates	Mar 7-9, 2015	May 15-Jun 15 2015	Jun 16-28, 2015
# sites	1K	10K	10K
# successfully loaded sites	959	8916	8731
# links extracted	8986	85049	72855

ate cookies. Unfortunately, this approach can become problematic if advertisements are also loaded in single iframes (rather than nested iframes), because a reload of an advertisement can cause an additional impression and double-billing for the advertiser.

Similarly, if a social widget is loaded in nested iframes, our heuristic would fail to recognize it and would not reload it. Fortunately, the effect would not be as bad as reloading advertisements, because the click will be just passed to the social widget, which may ask the user to login again. Nevertheless, if this case happens frequently, it can frustrate the user.

Here, we evaluate how prevalent these potentially problematic cases are today. For this purpose, we modified a popular Firefox add-on for developers, Firebug [19], and augmented it to record the encountered iframes on a page, including single and nested ones, along with their parent elements. For each element, we recorded their element attributes. For single iframes, we also recorded whether the iframe contained any hyperlinks that the user can click on. We then randomly selected 20K pages from our most recent crawl (i.e., Crawl 3). After manually logging into the OSN services, we visited the pages with the default policy of "accept all cookies". A total of 19462 pages were loaded successfully, so that we could record their iframes.

During our evaluation, we consider two additional properties of the iframes loaded on a page: visibility and reloadabilty.

Visibility. We first check whether the iframe is visible. If the iframe is not visible to the user, then the user cannot interact with such an iframe. As a result, our heuristic would never have to handle these cases.

We use three checks to determine the visibility of an iframe. First, we check the iframe's CSS style as well as the width and height attributes. For the CSS style, we look for 'display: none;' and 'visibility: hidden' property values. For the width and height values, we investigate the CSS style as well as the element attributes. If the width and height values are both set to be 0 or 1 pixels, or 0% of the respective dimension of the window, we consider the element to be invisible to the user.

Second, we check the position of the iframe. Specifically, we look for CSS attributes that indicate that the position of the iframe should be out of the current window boundaries. For this purpose, we conservatively look for 'position: absolute' property along with a negative pixel value for 'top', 'bottom', 'left' and 'right' positioning parameters (e.g., "position: absolute; top: -1000px"). Additionally, we use the stacking order of the elements via the 'z-index' property and consider a negative value as an indication that the iframe should be behind other elements, and thus, invisible.

Finally, we check the visibility of the iframe's parents in the DOM tree because an invisible parent will also cause the iframe to be invisible. We again check the same CSS properties and element attributes for each parent and mark the iframe invisible if any of the parents is invisible. For parent elements, we conservatively consider only the dimensions and display properties of the element (and not its position values).

Reloadability. After determining the iframe's visibility, we consider whether it can be reloaded by our heuristic if the user were to interact with it. If the iframe is not reloadable, then there is no point of considering this iframe within the context of our heuristic.

Table 2.2: Number of iframes recorded in successfully loaded 19462 pages in our random sample.

Type	Single	Nested
Invisible	57204	64613
Visible, Unreloadable	2121	6452
Visible, Reloadable, First-party	4073	12529
Visible, Reloadable, Third-party	19947	16647
Total	83345	100241

For this purpose, we check the source attribute (i.e., 'src') of the iframe. It is possible that the iframe does not have one, perhaps because it was generated via JavaScript. It is also possible that the source attribute contains some JavaScript code that executes when the iframe is first loaded by the browser. These cases are not considered by our heuristic and thus, are filtered from our results.

2.7.2.1 Advertisements in Single Iframes

We first investigate the single iframes that are directly included in the pages. In other words, these single iframes do not contain other iframes and are not contained within other iframes. Table 2.2 shows the breakdown of the iframes we recorded in our crawl. There were a total of 83345 single iframes. A big majority of these single iframes were invisible to the user. About 24% of them were visible, reloadable and belonging to a third-party domain. We consider these iframes for the evaluation of our heuristic.

We extract the source attributes of the iframes and match them to well-known social widgets. Any other third-party domain is conservatively considered an advertisement domain. These cases would constitute a false positive for our heuristic, because our heuristic would classify this iframe as a social widget and reload it instead of passing the click unchanged.

Table 2.3 shows the top 30 third-party domains ranked by the number of iframes. As

Table 2.3: Distribution by domain and category of single, third-party iframes that were visible and reloadable. Only top 30 domains with the most number of iframes are shown. The remaining sites (shown as 'Other domains') are conservatively assumed to be advertisement iframes.

Domain	Count (Percentage)	Category	Has cookies?
facebook.com	6921 (34.70%)	Social	Yes
twitter.com	3129 (15.69%)	Social	Yes
google.com	2752 (13.80%)	Social	Yes
doubleclick.net	1876 (9.41%)	Advertisement	Yes
youtube.com	1250 (6.27%)	Video/Social	Yes
disqus.com	505 (2.53%)	Social	Yes
exoclick.com	483 (2.42%)	Advertisement	Yes
stumbleupon.com	185 (0.93%)	Social	Yes
blogger.com	135 (0.67%)	Social	Yes
amazon-adsystem.com	122 (0.61%)	Advertisement	Yes
googleapis.com	114 (0.68%)	Advertisement	No
addthis.com	107 (0.54%)	Social	Yes
reddit.com	50 (0.25%)	Social	Yes
adnxs.com	49 (0.25%)	Advertisement	Yes
sitescoutadserver.com	38 (0.19%)	Advertisement	Yes
lockerdome.com	38 (0.19%)	Social	Yes
juicyads.com	37 (0.19%)	Advertisement	Yes
2mdn.net	37 (0.19%)	Advertisement	No
yimg.com	37 (0.19%)	Advertisement	Yes
shopifyapps.com	37 (0.19%)	Advertisement	Yes
eblastengine.com	34 (0.17%)	Advertisement	No
springboardplatform.com	32 (0.16%)	Advertisement	Yes
adblade.com	30 (0.15%)	Advertisement	Yes
wistia.net	26 (0.13%)	Video	Yes
zedo.com	24 (0.12%)	Advertisement	Yes
redditstatic.com	24 (0.12%)	Social	Yes
tumblr.com	23 (0.12%)	Social	Yes
tout.com	23 (0.12%)	Video	Yes
youtube-nocookie.com	21 (0.11%)	Video/Social	No
myvoicenation.com	21 (0.11%)	Social	Yes
Other domains (465)	1894 (9.50%)	Advertisement*	Yes (299), 157 (No)
All domains	19947 (100%)	-	-

one can see, the majority of the single iframes belongs to social widgets by Facebook, Twitter and Google+. These widgets constitute about 65.83% of the single iframes. A further 12.07% belong to video sites that allow embedding of videos (e.g., Youtube), commenting platforms (e.g., Disqus), blog sites with social interactions (e.g., Blogger, Stumbleupon) and smaller social networks (e.g., Addthis, LockerDome).

The biggest false positive with 9.41% is the doubleclick.net iframes. Upon further inspection, we find out that 1646 (87.74%) of these 1876 iframes are called 'view-through conversion' iframes. When we manually visited some of the pages that included these iframes, we noticed that all of these iframes are transparent, such that they take the embedding page's background color. Furthermore, these iframes did not contain any links nor any content, and were located at the bottom of the viewed page. Even though our heuristic would reload these iframes if clicked by the user, the likelihood of the user click seems quite low. A further 122 (6.50%) did not contain any links. Most advertisements contain either an image or some descriptive text about the advertisement, which links to the landing page of the advertiser. As a result, the likelihood that these iframes were used to show an advertisement seems low, which in turn reduces the likelihood of the user interacting with this iframe. Out of the remaining 108 iframes, 40 of these iframes by doubleclick.net were present only in parked domains, which showed only advertisements.

The second biggest false positive with 2.42% is iframes by exoclick.com. When inspected, we found that these iframes only existed in adult sites. There were 59 such domains and 145 pages on these domains. We also checked the other iframe domains that were present in these same pages. Out of 590 single (visible and reloadable) iframes found, there were a total of 47 social widgets belonging to Google+ (20), Twitter (15) and Facebook (12). The rest of the iframes belonged to other adult sites.[3]

[3]There were no social widgets present in the (visible and reloadable) 427 nested iframes.

For the remaining single iframes (shown as 'Other' domains in Table 2.3), we assume conservatively that they all belong to the advertisement category. The total number for advertisement iframes, including the 108 doubleclick.net and 483 exoclick.com iframes, is 3076. This number corresponds to a false positive rate of 15.42%, in which our heuristic would recognize an advertisement as a social widget and reload it.

During our inspection, we notice that some of these third parties do not set any cookies. To understand how frequent this case is, we check the cookie database of the Firefox profile. We find that 307 single iframes were from domains that did not contain any links and did not have any cookies. Such domains can be content distribution network sites or cloud operators, which are only used to serve advertisement files such as banners, images and JavaScript libraries. Examples include 'googleapis.com', and '2mdn.net'. In these cases, our heuristic would not reload the iframe: The click would not find a target link, and thus, use the iframe's source. Because the iframe domain did not have any cookies, the click would be passed without reloading the iframe. Compensating for these cases, our heuristic's false positive rate drops to 13.88%.

In iframes that contained links, it is also possible that the target link belongs to a domain that did not set any cookies. In these cases, the iframe will not be reloaded either. We find that the number of links belonging to domains with cookies and without cookies are almost equal, constituting 41.03% and 41.28% of the links, respectively.[4] As a result, it is still possible that our heuristic would pass the click without reloading the iframe.

2.7.2.2 Social Widgets in Multiple Iframes

Next, we check the nested iframes. A nested iframe is an iframe that either contains a child iframe or that is contained by another iframe. Table 2.2 shows the total numbers of nested iframes. Again, a big majority of these iframes are invisible to the user. There

[4]The rest were either first party links or non-href (e.g., 'javascript:' links).

were 16647 nested iframes that can be considered for our heuristic's evaluation.

For our evaluation, we investigate the source attributes of these iframes. If any of these sources belong to a social widget, we consider that as a failure of our heuristic.

Table 2.4 shows the top 30 third-party domains ranked by the number of iframes. As one can see, the significant majority of these third parties belong to the advertisement category. For these iframes, our heuristic would correctly pass the click without reloading the iframes.

Table 2.5 shows the number of nested iframes containing a social widget. There were a total of 679 iframes constituting about 4.07% of the total. These cases are false negatives for our heuristic, because it would fail to recognize these widgets. When the click is passed through, the widget might ask the user to log in again instead of being reloaded with cookies.

2.7.2.3 Distribution of Social Widgets

We then investigate the distribution of social widgets. Table 2.6 shows the numbers (and percentages) of social widgets present in single or nested iframes. Again, the majority of social widgets were present in single iframes (i.e., 93.79%), including widgets belonging to bigger online social network domains, such as facebook.com, twitter.com and google.com.

2.7.2.4 Distribution of Advertisements

We also investigate the distribution of advertisement iframes. We conservatively consider every domain that is not present in Table 2.6 as an advertisement domain. We again remove the false positives associated with doubleclick.net (i.e., 1646 'view-through conversion' iframes and 122 iframes with no links) that have been explained in Section

Table 2.4: Distribution by domain and category of nested, third-party iframes that were visible and reloadable. Only top 30 domains with the most number of iframes are shown.

Domain	Count (Percentage)	Category	Has cookies?
doubleclick.net	7162 (43.02%)	Advertisement	Yes
googlesyndication.com	2141 (12.86%)	Advertisement	Yes
adnxs.com	1023 (6.15%)	Advertisement	Yes
criteo.com	672 (4.04%)	Advertisement	Yes
2mdn.net	389 (2.34%)	Advertisement	No
exoclick.com	302 (1.81%)	Advertisement	Yes
brandwire.tv	274 (1.65%)	Advertisement	Yes
yummly.com	177 (1.06%)	Social	Yes
vimeocdn.com	172 (1.03%)	Video	No
mediaplex.com	151 (0.91%)	Advertisement	Yes
vimeo.com	147 (0.88%)	Video	Yes
google.com	144 (0.87%)	Social	Yes
amazon-adsystem.com	136 (0.82%)	Advertisement	Yes
teleskipp.de	134 (0.80%)	Video	No
veruta.com	124 (0.75%)	Advertisement	Yes
optmd.com	99 (0.59%)	Advertisement	Yes
adspirit.de	91 (0.55%)	Advertisement	Yes
rfihub.com	80 (0.48%)	Unknown	Yes
facebook.com	79 (0.47%)	Social	Yes
ad4mat.de	72 (0.43%)	Advertisement	No
truste.com	66 (0.40%)	Advertisement	Yes
w55c.net	65 (0.40%)	Advertisement	Yes
ero-advertising.com	62 (0.37%)	Advertisement	Yes
youtube.com	62 (0.37%)	Video/Social	Yes
instagram.com	56 (0.34%)	Social	Yes
solocpm.com	55 (0.33%)	Advertisement	Yes
tacdn.com	51 (0.31%)	Unknown	Yes
pubmatic.com	51 (0.31%)	Advertisement/Analytics	Yes
ibm.com	48 (0.29%)	Unknown	Yes
ato.mx	48 (0.29%)	Advertisement	Yes
Other domains	2514 (15.10%)	Mixed	Yes (295), No (128)
All domains	16647 (100%)	-	-

Table 2.5: Number of nested iframes belonging to an online social network widget (i.e., false negatives).

Domain	Count (Percentage)
yummly.com	177 (1.06%)
google.com	144 (0.87%)
facebook.com	79 (0.47%)
youtube.com	62 (0.37%)
instagram.com	56 (0.34%)
lockerdome.com	47 (0.28%)
blogger.com	46 (0.28%)
linkedin.com	29 (0.17%)
disqus.com	19 (0.11%)
twitter.com	13 (0.08%)
massrel.io	7 (0.04%)
Total	679 (4.07%)

2.7.2.1. Table 2.7 shows the number (and percentages) of these advertisements present in single or nested iframes. As one can see, the majority of advertisements (90.49%) were served in nested iframes.

2.7.2.5 Summary

We can summarize our results as follows. The majority of advertisements are placed in nested iframes. Similarly, the majority of social widgets are present in single iframes. We find that the practice of placing advertisements in single iframes and social widgets in nested iframes is not prevalent in the Internet today. We conclude that our heuristic would work as desired. Section 2.8.1 discusses the robustness of our heuristic to changes.

2.7.3 Performance Overhead

To have a better understanding of our policy's performance, we recorded the page load times of existing cookie policies as well as our policy. To do so, we created three more Firefox profiles, in addition to the default policy of "accept all cookies" we used to

Table 2.6: Distribution of iframes from social widget domains.

Domain	Total # iframes from domain	# single iframes (Percentage)	# nested iframes (Percentage)
facebook.com	7000	6921 (98.87%)	79 (1.13%)
twitter.com	3142	3129 (99.59%)	13 (0.41%)
google.com	2896	2752 (95.03%)	144 (4.97%)
youtube.com	1312	1250 (95.27%)	62 (4.73%)
disqus.com	524	505 (96.37%)	19 (3.63%)
stumbleupon.com	185	185 (100.0%)	0 (0.0%)
blogger.com	181	135 (74.59%)	46 (25.41%)
yummly.com	177	0 (0.0%)	177 (100.0%)
vimeocdn.com	174	2 (1.15%)	172 (98.85%)
vimeo.com	148	1 (0.68%)	147 (99.32%)
addthis.com	108	107 (99.07%)	1 (0.93%)
lockerdome.com	85	38 (44.71%)	47 (55.30%)
instagram.com	56	0 (0.0%)	56 (100.0%)
reddit.com	50	50 (100.0%)	0 (0.0%)
wistia.net	33	26 (78.79%)	7 (21.21%)
linkedin.com	29	0 (0.0%)	29 (100.0%)
redditstatic.com	24	24 (100.0%)	0 (0.0%)
tout.com	23	23 (100.0%)	0 (0.0%)
tumblr.com	23	23 (100.0%)	0 (0.0%)
youtube-nocookie.com	21	21 (100.0%)	0 (0.0%)
massrel.io	7	0 (0.0%)	7 (100.0%)
All domains	16198	15192 (93.79%)	1006 (6.21%)

Table 2.7: Distribution of iframes from advertisement domains. These domains are the remaining domains after the social widget domains (shown in Table 2.6) are removed from the list of all domains with nested iframes. Only top 30 domains with the most number of iframes are shown.

Domain	Total # iframes from domain	# single iframes (Percentage)	# nested iframe (Percentage)
doubleclick.net	7270	108 (1.49%)	7162 (98.51%)
googlesyndication.com	2147	6 (0.28%)	2141 (99.72%)
adnxs.com	1072	49 (4.57%)	1023 (95.43%)
exoclick.com	785	483 (61.53%)	302 (38.47%)
criteo.com	673	1 (0.15%)	672 (99.85%)
2mdn.net	426	37 (8.69%)	389 (91.31%)
brandwire.tv	274	0 (0.0%)	274 (100.0%)
amazon-adsystem.com	258	122 (47.29%)	136 (52.71%)
mediaplex.com	164	13 (7.93%)	151 (92.07%)
teleskipp.de	134	0 (0.0%)	134 (100.0%)
googleapis.com	128	114 (89.06%)	14 (10.94%)
veruta.com	124	0 (0.0%)	124 (100.0%)
optmd.com	105	6 (5.71%)	99 (94.29%)
adspirit.de	91	0 (0.0%)	91 (100.0%)
rfihub.com	85	5 (5.88%)	80 (94.12%)
ad4mat.de	72	0 (0.0%)	72 (100.0%)
ero-advertising.com	72	10 (13.89%)	62 (86.11%)
truste.com	66	0 (0.0%)	66 (100.0%)
w55c.net	65	0 (0.0%)	65 (100.0%)
everesttech.net	57	16 (28.07%)	41 (71.93%)
solocpm.com	55	0 (0.0%)	55 (100.0%)
tacdn.com	51	0 (0.0%)	51 (100.0%)
pubmatic.com	51	0 (0.0%)	51 (100.0%)
cloudfront.net	49	17 (34.69%)	32 (65.31%)
ibm.com	48	0 (0.0%)	48 (100.0%)
ato.mx	48	0 (0.0%)	48 (100.0%)
adxpansion.com	47	0 (0.0%)	47 (100.0%)
yimg.com	46	37 (80.43%)	9 (19.57%)
adblade.com	45	30 (66.67%)	15 (33.33%)
springboardplatform.com	42	32 (76.19%)	10 (23.81%)
Other domains	2699	554 (20.53%)	2145 (79.47%)
All domains	17249	1640 (9.51%)	15609 (90.49%)

Table 2.8: Number of successfully loaded links.

	Crawl 1	Crawl 2	Crawl 3
Accept all cookies	8654	81466	69798
Accept third party cookies from visited	8519	81798	69912
Never accept third party cookies	7975	81888	69913
Our policy (Priv3+)	8538	81859	69953
Loaded by all	7257	73835	63567

collect the links. Two of these profiles correspond to each of the remaining cookie policy settings, namely "accept third party cookies from visited" and "never accept third party cookies". Similar to the "accept all cookies" profile, we installed PageRecorder add-on for these profiles. For the last Firefox profile, we modified our Priv3+ add-on to augment it with the same functionality to record the URL of the page and the page load times.

We again waited 45 seconds for a page to successfully load. Nevertheless, some pages were not successfully loaded within the allowed time. Furthermore, due to fluctuations in network and server conditions, not all pages were loaded successfully with all of our Firefox profiles. For example, it is possible that the profile with "accept all cookies" policy was able to load a page successfully, but the profile "accept all cookies from visited" was not. Nevertheless, there was a substantial number of links that were loaded successfully by all four profiles. Table 2.8 shows the number of successfully loaded links.

Table 2.9 shows the average page load times as well as the relative overhead. We consider the profile with the "accept all cookies" policy as the baseline. As one can see, our policy does not have a significant performance impact on page load times.

Table 2.9: Performance overhead of cookie policies for links successfully loaded by all four profiles.

	Crawl 1	Crawl 2	Crawl 3
Accept all cookies	4617.66ms (-)	3343.37ms (-)	3040.97ms (-)
Accept third party cookies from visited	4501.28ms (1.32%)	3329.86ms (-0.40%)	3140.19ms (3.26%)
Never accept third party cookies	4519.64ms (1.73%)	3313.97ms (-0.88%)	3136.72ms (3.15%)
Our policy (Priv3+)	4617.66ms (3.94%)	3445.91ms (3.07%)	3106.58ms (2.16%)

2.8 Discussion

2.8.1 Robustness of the Heuristic

A natural question to ask at this point is whether our heuristic is robust to changes in the way social widgets and advertisements are used on web pages. The following two cases are possible.

In the first case, the OSN providers would start embedding their social widgets in nested iframes to evade our heuristic. With our heuristic, these social widgets will never be reloaded with user cookies when interacted with; the click will be just passed through. As a result, the social widget will have to prompt the user to login again, inconveniencing the user. Making it more difficult for the users to interact with social widgets only hurts an OSN provider's ability to obtain user behavior and provide a better user experience as well as monetize its services. For this reason, there is no incentive for OSN providers to provide their social widgets in nested iframes.

In the second case, the advertisements would be placed in single iframes to make our heuristic fail to detect them. With our heuristic, clicks to an advertisement will not be passed through, but the advertisement iframe will be reloaded. This action does not benefit the advertisement network nor the advertiser for three reasons: First, the advertiser would be losing a potential customer, resulting also in reduced revenue for the advertisement network. Second, the reloading would cause additional impressions and charges for the advertiser, in which case the advertisement network would have to resolve this issue with additional cost. Finally, although reloading with cookies would send the user cookies to the advertisement network or advertiser, any attempt to track the users will not be successful: when the user visits another page containing advertisements from this advertisement network or advertiser, those advertisements will also be loaded without sending user cookies. For these reasons, there is no incentive for the advertisement networks or the advertisers to place advertisements in single iframes.

These two cases suggest a lack of incentives for the OSN providers and advertisement networks to evade our heuristic. In fact, there are incentives for them to place the social widgets in single iframes and advertisements in nested iframes, respectively, such that they can benefit from any user interaction with third-party content present in these iframes as much as possible. Consequently, our heuristic will have fewer false positives and false negatives.

2.8.2 Reloading Advertisement Iframes with Cookies

One can argue that a clicked advertisement shows the user's intention to interact with the advertiser, and thus, the advertisement iframe can be reloaded. This approach would also alleviate the problems associated with the accuracy of our heuristics. Although this action sounds plausible, this approach has the following problems: First, it is not clear which iframe to reload because there may be multiple, nested iframes. Reloading the parent may generate a different child iframe containing another advertisement less relevant to the user. Similarly, reloading the child iframe may not end up producing a more relevant advertisement. Even if this decision can be made, some iframes do not have their 'src' property set preventing the reload. Most importantly, reloading such iframes may have adverse effects for the advertisers, triggering new auctions and double-charging for impressions and clicks. For these reasons, we pass the click to an advertisement unchanged.

2.8.3 Evercookies

Evercookies use storage vectors (e.g., Flash cookie store) [76], which are not deleted when the browser cookies are cleared. Trackers exploit these storage vectors to respawn old cookie values to achieve a longer persisting tracking period. Our policy would prevent these respawned cookies to be sent to third parties unless there is user interaction.

2.8.4 Cookie Synching

Our policy partially prevents cookie synching that uses *previously* set third-party cookies. The prevention of third-party scripts to access cookie values (Section 2.5.6) may also prevent other methods like using first-party cookie values as parameters for third-party resources.

2.8.5 Behavioral Advertisements & Extended Web Analytics

Our policy prevents third-party tracking used for behavioral advertisements, which may be deemed necessary for a sustainable web. A byproduct of this tracking is extended web analytics, in which the aggregators can provide visitor demographics. There have been multiple efforts to provide behavioral advertising that is privacy-preserving and comparable to today's systems [115,143,151]. Similarly, previous research, as well as our system described in the next chapter, shows how the same aggregate information can be obtained without violating user privacy [97,98]. We think these efforts as well as our policy are steps in the right direction to provide essential services for a sustainable web without compromising user privacy.

2.9 Conclusion & Future Work

We proposed and explored a general and interaction-based third-party cookie policy. With our policy, third party content is loaded without sending associated third-party cookies, effectively preventing tracking by OSNs, data aggregators and behavioral advertisers. This policy strikes a balance between functionality of social networking and privacy by requiring user interaction to reload the social widgets with cookies when the user wants. Our policy is general and does not depend on a blacklist, automatically solving problems associated with maintenance, distribution and circumvention of the blacklist. Finally, it supports non-tracking analytics and advertisement services, and does not penalize publishers who use these more privacy-friendly tools.

We have evaluated our policy on web pages from popular websites. According to our findings, our policy would work well in distinguishing social widgets and advertisements, such that social widgets can be activated on-demand, with low false positives and false negatives. While doing so, our policy does not impose a significant performance overhead on page load times. Furthermore, our heuristic is robust to changes in the way social widgets and advertisements are loaded, because there are no incentives for OSN providers and advertisers to cheat our heuristic.

In the future, we hope to gather more users and obtain their feedback. Such feedback will help us better understand how users perceive and treat different types of third-party content. Ideally, we would like our policy to be implemented and supported in major browsers, such as Mozilla Firefox, Google Chrome and Apple Safari. In the long run, we hope that third-party service providers for advertisements, analytics and social widgets will learn to respect user privacy by not requiring cookie values until the user interacts with the third-party content.

As discussed in Section 2.8.5, our policy affects analytics services that depend on third-party tracking. In the next chapter, we describe a web analytics system that can provide (at least) the same amount of aggregate analytics information about visitors to publishers without tracking users.

Chapter 3

Non-tracking Web Analytics

In this chapter, we present the design, implementation and analysis of a practical, privacy-preserving and non-tracking web analytics system. Our system enables a website publisher to directly query its users for extended web analytics information by acting as an anonymizing proxy between the clients and the data aggregator. As a result, no new system components are introduced into the ecosystem, making the system more easily adoptable. The users also benefit from this arrangement, because they are given anonymity and unlinkability properties via differential privacy mechanisms while providing potentially sensitive data for analytics. This work was published in the Proceedings of the ACM Conference on Computer and Communications Security (CCS) in 2012 [81].

This chapter is organized as follows. The next section introduces the privacy problem by third-party tracking in the web analytics ecosystem and motivates the need for a non-tracking alternative. Section 3.2 presents our contributions to provide this alternative approach. In Section 3.3, we describe the components existing in today's systems as well as in our system. Sections 3.4 and 3.5 list our functionality as well as privacy goals and our assumptions, respectively. Section 3.6 presents an overview of our system.

In Section 3.7, we describe our system in detail. Section 3.8 presents an informal but detailed analysis of our system. Our implementation and evaluation are presented in Section 3.9. We conclude in Section 3.10.

3.1 Introduction

Website publishers use web analytics information to analyze their traffic and optimize their site's content accordingly. Publishers can obtain analytics data by running their own web analytics software programs [7, 40, 41]. These analytics programs provide publishers with statistics about users on their site, such as page views, clickstreams, browsers, operating systems, plugins as well as frequency of returning visitors. However, they do not provide other potentially useful information, such as user demographics.

For this reason, publishers often outsource the collection of web analytics to a third party data aggregator, such as comScore, Quantcast or StatCounter [6, 62]. A data aggregator collects data from users visiting a publisher's website and presents these data in aggregate form to the publisher. Besides being convenient for publishers, because they only have to embed a small piece of code (i.e., a JavaScript code snippet), this outsourcing allows publishers to learn statistical information they could not otherwise learn from their own web server logs, such as the demographics of their users and other websites their users visit. A data aggregator can *infer* this *extended web analytics* information because it collects user data across many publisher websites. Compiling extended web analytics via these collected data also benefits the data aggregator because it can sell this information to advertisers and publishers alike.

Although this scheme is beneficial for the publishers and the data aggregators, it raises concerns about users being tracked while browsing the web. This tracking enables a data aggregator to compile detailed behavior of individual users, and infer individual user demographics [144]. Thus, data aggregators are given a lot of information about users' actions on the web and have to be trusted that they will not abuse it. This trust has been violated in the past [33,47,49].

Tracking affects not only users, but also the data aggregators themselves, who are often criticized for this behavior. These criticisms have led to industry self-regulation to provide opt-out mechanisms [10,11,50,66], the Do-Not-Track (DNT) initiative in the W3C, and many client-side tools, either to implement DNT [24,52,57], or to prevent tracking outright [15,23,29,70]. To the extent that these efforts take hold, the ability for data aggregators to provide extended analytics to publishers will be degraded.

In addition, even with tracking, inferring accurate user demographics is a difficult task that may produce inconsistent results. In 2012, we found examples of such inconsistencies among the biggest data aggregators, Quantcast and Doubleclick. According to Quantcast, 24% of rottentomatoes.com's visitors in US were between 18 and 24, and 20% were between 35 and 44. On the other hand, according to Doubleclick, these numbers were 10% and 36%, respectively. Similar inconsistencies also existed for other sites. Doubleclick has since discontinued their reporting of audiences for advertisement purposes, whereas Quantcast stopped publishing audience data for sites that do not utilize Quantcast for advertisement/analytics purposes.

In a more recent comparison between Alexa and Quantcast, we found the following inconsistencies: grindtv.com, an extreme sports and entertainment site, is using Quantcast. Quantcast's measurement states that 67% of the visitors are male [27]. On the other hand, Alexa estimates that male and female visitors are almost equally distributed [26]. Similarly, usnews.com, another Quantcast using site, is reported to have a gender distribution at 55% females and 45% males [65], whereas Alexa estimates that the males are greatly under-represented on this site [64].

These examples show that tracking users and inferring results from collected information may not be the best method for obtaining accurate extended web analytics (i.e., user demographics). An alternative (e.g., direct querying of user data) might be better suited for this purpose, provided that user privacy is protected via anonymity.

3.2 Contributions

To address the above issues, we present the design and implementation of a practical, privacy-preserving and non-tracking web analytics system. In our system, user information is stored in a database on the user device (client). We exploit the direct communication that naturally takes place between the publisher and the users during a page visit by having the publisher distribute database queries to clients, and by having the publisher act as an anonymizing proxy for the (encrypted) answers from the clients to the data aggregator. The aggregator aggregates the anonymous answers, and provides the aggregate results to the publisher. Both the publisher and the aggregator add differentially-private noise before passing data on to each other.

To the best of our knowledge, we are the first to study the problem of collecting extended web analytics information without tracking users. We describe and analyze a novel design that provides the first practical solution to this problem. Our solution eliminates the need for third-party tracking for extended web analytics purposes, does not require new organizational players, and is practical to deploy.

Keeping user data at the client device and utilizing the publisher as a proxy between the clients and the aggregator, our system allows publishers to *directly query* extended web analytics rather than rely on inferred data. Combined with the differentially-private noise mechanisms, our protocol enables aggregation of users' private information, such as demographics and websites visited, without violating individual user privacy (see 3.4.2 for details) under a set of realistic threat assumptions. As a result, the system can provide better analytics than current services, in terms accuracy and variety.

While the decision to use the publisher as a proxy is good for deployability, it creates new technical challenges because publishers can be malicious. In particular, they might try to exploit their position in the middle by manipulating which clients receive and answer queries, and to overcome the noise added by the aggregator using repeated queries. Our system has mechanisms to raise the bar for such publishers and render these attempts more difficult.

We implemented and evaluated our system to gauge its feasibility. We report on performance benchmarks and describe our deployment across several hundred users.

3.3 Definitions & Components

We define **extended web analytics** as any additional information that the publisher cannot obtain by investigating her own web server logs. Extended web analytics may contain demographics (e.g., age, gender, education level, income, marital status) and web browsing habits (e.g., other websites a user visits, search phrases used on search engines, products viewed on shopping sites) as well as any other information regarding the user's environment (e.g., CPU load while viewing certain pages, applications installed).

There are three entities in today's tracking web analytics systems: the publisher, the data aggregator, and the client. Publishers create websites. Data aggregators provide publishers with aggregation service for web analytics. Users use their clients (e.g., the browser) to access and consume the content that publishers host.

Figure 3.1 shows the interactions between these entities today. When clients visit the publisher's website (step 0), they also send analytics data to the data aggregator via the code snippet installed on the publisher's website (step 1). After collecting information from individual clients, the data aggregator aggregates analytics information (step 2),

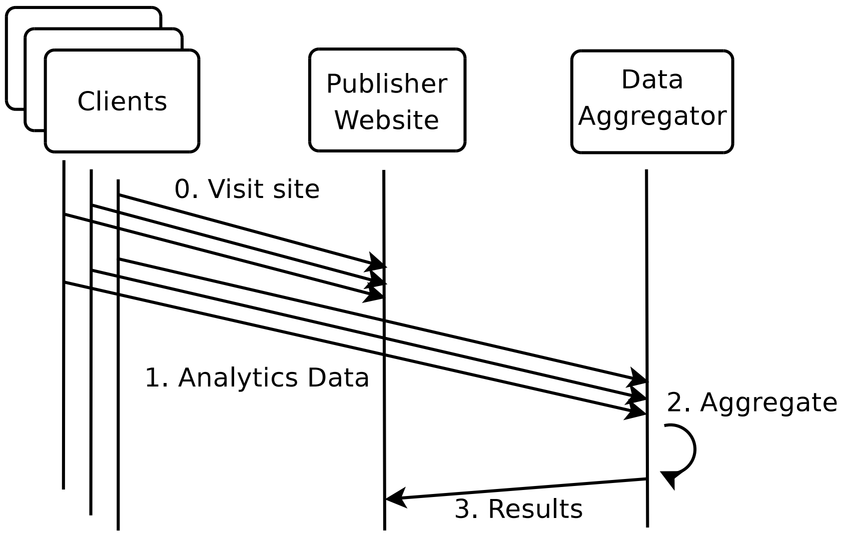


Figure 3.1: Operation of today's tracking web analytics systems.

during which the aggregator infers extended web analytics information. The aggregator then shares aggregate result with the publisher (step 3).

Our system consists of the same components. However, as we describe later, our protocol eliminates the need for tracking to obtain extended web analytics.

3.4 Goals

3.4.1 Functionality Goals

We would like our system to provide publishers and data aggregators with *at least* the same aggregate information they obtain in today's systems. More specifically, publishers should get more accurate and more types of web analytics information than they do today if possible. Data aggregators should also obtain web analytics information for all of their partner publishers like they do today, as an incentive for performing aggregation. Note that today most data aggregators provide behavioral advertising, for which they require individual user information, and not aggregate data. Given that other research shows how to accomplish behavioral advertising without exposing user information [115,151], we assume that the data aggregator requires only aggregate data.

We would also like to avoid requiring new players like proxies. While potentially useful for the operation of the system, additional players can also hinder adoption and practicality.

Ideally, we would want our system be more efficient than today's systems, it might not also be possible given our other functionality and privacy goals. As a result, our system should scale at least adequately.

Finally, the system should not allow clients or publishers to manipulate results beyond what is possible today (i.e., via botnets).

3.4.2 Privacy Goals

Our main user privacy goal is to eliminate third-party tracking of users across the web. To this end, we want to provide the visitors with anonymity from the data aggregator. Today, unless a visitor uses a proxy, she is exposed to the aggregator via her network address.

The network address is, however, not the only identifier the aggregator can use to track a visitor across the web. Unique pieces of information about a visitor, either individually or as a combination, can enable the aggregator to identify a user. Although the visitor may have anonymity through other means (e.g., proxy), it may still be possible for the aggregator to anonymously profile a visitor. To mitigate this problem, our system should ensure that the information obtained by the aggregator is unlinkable.

Our system should also give users information about their privacy loss with respect to each publisher and each data aggregator. Such information is available within the formal guarantees of differential privacy (DP) [104]. Besides DP's privacy loss concept, its noise mechanism also helps us to achieve our unlinkability goal.

Privacy Non-goals. While each client in our system knows about its privacy loss, it should be noted that such knowledge at the client is of limited value. DP is very conservative, because it assumes that the attacker may have arbitrary auxiliary information it can use to discover information about users in the database. When the attacker does not have this auxiliary information, which is the common case, DP's measure of privacy loss is overly pessimistic. Although a client in our system could, in theory, refuse to answer queries if a privacy budget is exceeded, doing so is not practical in our setting, because a query may be legitimately repeated from time to time (e.g., to measure changes in the user base). Furthermore, DP's privacy loss measure assumes a static database, whereas in our setting, the "database" is dynamic: the user population for a given publisher changes almost constantly, and some individual user data may change as time passes.

For these reasons, it is unrealistic, and in our setting, unnecessary to set a hard limit on user privacy loss (i.e., budget). In this regard, we do not aim to provide users with formal DP guarantees. Nevertheless, we find DP to be a valuable mechanism, in part because it provides a worst-case measure of privacy loss, but primarily because the noise added to answers substantially raises the bar for the attacker, while still providing adequate accuracy for aggregate results.

3.5 Assumptions

3.5.1 Client

We assume that the users trust the client software, in terms of the data it stores locally and its operation, just as they trust their browser software today. While it is possible for a browser to be infected by malware, such malware is in a position to violate user privacy in many ways beyond our system; thus, we do not protect against this threat.

By contrast, we assume that the clients may be malicious towards the publisher and the data aggregator. A malicious client may attempt to distort aggregate results, similar to the situation today where a client may, for instance, participate in click fraud. It may also try to violate the privacy of other users, possibly colluding with the publisher.

3.5.2 Data Aggregator

We assume that the data aggregator is honest-but-curious (HbC); in other words, that it obeys the prescribed operation, but may try to exploit any information learned in the process. As an HbC player, we assume that the data aggregator does not collude with the publishers. In principle, a malicious publisher could of course simply choose to work with a malicious data aggregator. We assume a setup whereby aggregators state their non-collusion in a privacy statement, making them legally liable and subject to punishment (e.g., by the FTC). An aggregator that is also a publisher would have to internally separate information.

We justify such an HbC aggregator on the assumption that the client software plays an overseer role, and allows only HbC aggregators to participate. For instance, the browser could refuse inclusion to any aggregator that does not provide such a privacy statement, or appears untrustworthy for any other reason. Today, browsers already play a similar role in a number of respects, such as by selecting default certificate authorities and search engines, and in some cases, by warning users of potentially harmful websites. In today's industry setting where major data aggregators can generally be expected to operate within the legal framework of their own stated privacy policies, we think that this assumption is reasonable.

As stated earlier in our goals, we assume that the data aggregator requires only aggregate data in the context of web analytics. We leave behavioral advertising outside our scope and refer the readers to other research [115, 151].

3.5.3 Publisher

We assume that the publisher is selfishly malicious both towards the users and the data aggregator, meaning that the publisher will try to only benefit itself. As a potentially malicious player, the publisher may try to violate the privacy of users with correct clients. In particular, because the publisher distributes queries and collects answers, it is in a position to selectively query clients, drop selected client answers, and add answers beyond those required for DP noise. This position leads, for instance, to an attack whereby the publisher *isolates* a single client by dropping all answers except for those of the single client, and providing fake answers instead of the dropped answers. With repeated queries to such an isolated client, the publisher may overcome the added DP noise. The publisher may also be motivated to falsify the results it gives to the data aggregator, for instance, to appear more popular or more attractive to advertisers. Our design has mechanisms to mitigate the effect of these behaviors. Note, however, that we assume that the publisher correctly adds DP noise to answers, because withholding noise does not benefit the publisher, and the minor reduction in overhead gained (Section 3.9) is not adequate incentive.

Publishers today can directly measure user activity on their websites (e.g., pages visited, links clicked). In addition, websites can often legitimately obtain additional information directly from users, such as personally identifiable information (PII), shopping activity, friends and hobbies. Information obtained directly from users by publishers is considered outside our scope.

3.5.4 Incentives

The incentives for the publisher and the aggregator are that they can obtain more accurate web analytics, because we directly measure attributes rather than infer them. Furthermore, they can get more types of web analytics that are not available today, such as how many pages users visit in a certain period or search engines used by the users of a publisher (Section 3.9). We do not think that users are incentivized. Although publishers could offer incentives to users (e.g., better content for participating users) to create an incremental deployment environment, we think that the browser is a better option for deployment. These entities (i.e., publishers, aggregators, and browsers) should also be motivated to provide better privacy to users. Even though we do not know for certain whether our stated incentives are adequate, we think that they are at least feasible.

3.6 System Overview

Our system comprises the same three entities that exist today: the client, the publisher, and the data aggregator (Figure 3.2). The publisher plays an expanded role: it distributes queries to clients, and it proxies client-aggregator communication. This role requires the publisher, or its hosting center, to install new software. While this requirement reduces ease-of-use compared to today, we think it is reasonable: many publishers already run their own analytics software [6,7,40,41,62] and hosting companies already offer servers with web analytics software pre-installed [18,31,42].

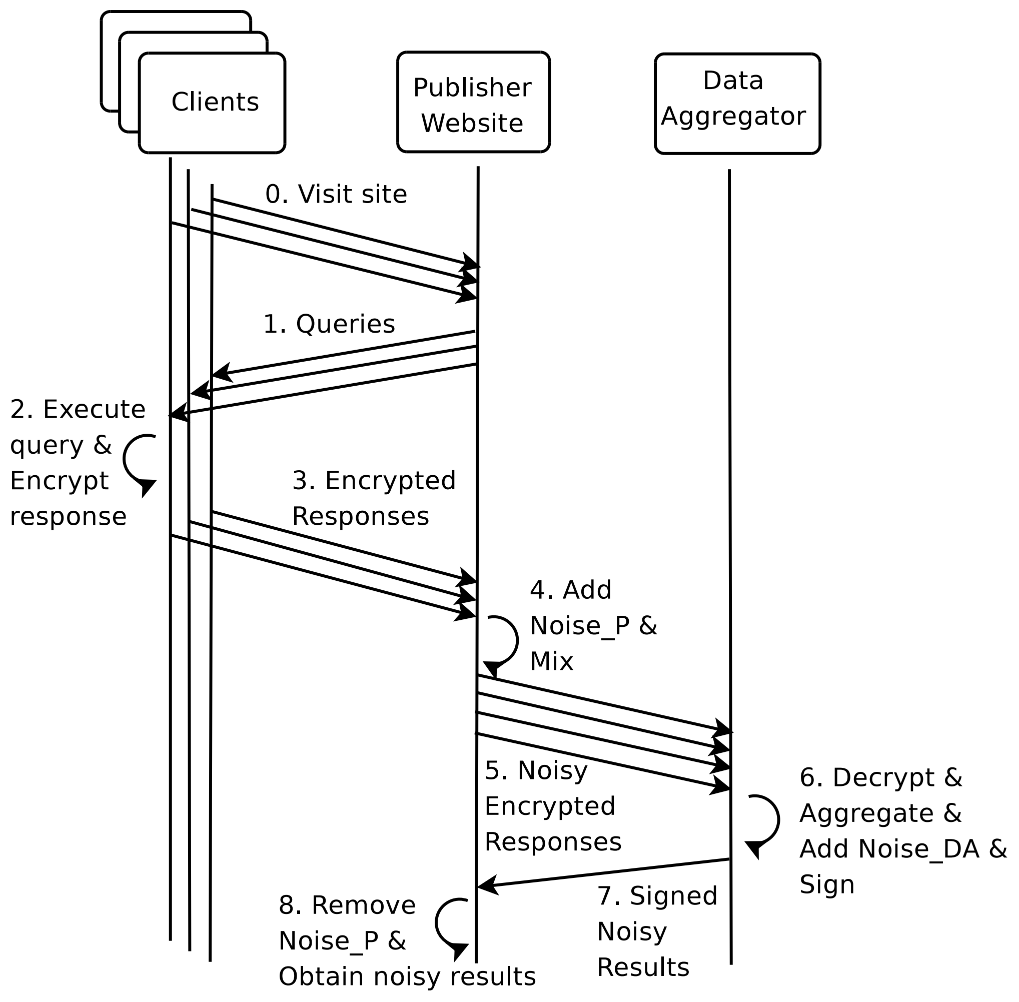
The client gathers and stores user information in a local database. Using this local database, the client requests and answers publisher queries when the user visits publisher sites. While the operation of the client is automatic, the user can always stop the client from gathering information or answering publisher queries.

The information collected and stored by the client can consist of extended web analytics (e.g., demographics, browsing behavior). We envision that the client scrapes most of this information from web pages the user visits (with informed user consent), such as online social networks, shopping websites and search engines, or the client can infer some information, like income. This scraping functionality can be supported by the browser. A recent and similar example is the Firefox User Personalization project [20,63]. The browser may also implement basic messaging, encryption and database mechanisms, and provide a sandboxed plugin environment for clients from different aggregators. The user can also provide some information directly.

To distribute queries to clients, publishers post queries at well-known URLs on their websites. When clients visit a website (step 0 in Figure 3.2), they download and read the queries (step 1).

Queries may be formulated by both the publisher and the data aggregator. While the queries themselves may be quite complex (i.e., SQL), the answers are limited to 'yes' and 'no' values. For instance, for the age distribution of users, the query effectively asks clients to evaluate 'yes' or 'no' for each age range of interest (e.g., <18, 18-34, 35-50, >50). This answering mechanism is achieved by defining *buckets*, such that each bucket corresponds to a potential answer value, and by mapping the query result to these buckets. Ultimately, the aggregator generates a per-bucket histogram of user counts. One benefit of using such bucket definitions is to limit the distortion a malicious client can impose on the aggregate result.

Each generated answer is separately encrypted with the public key of the data aggregator (step 2). Queries may have thousands of defined buckets, most of which have 'no' answers; for instance, one for each website a user may visit, or for each interest a user may have. To reduce the number of cryptographic operations, 'no' answers are omitted at the client. Instead, clients generate a specified number of answers which are



Clients
Publisher Website
Data Aggregator
0. Visit site
1. Queries
2. Execute query & Encrypt response
3. Encrypted Responses
4. Add Noise_P & Mix
5. Noisy Encrypted Responses
6. Decrypt & Aggregate & Add Noise_DA & Sign
7. Signed Noisy Results
8. Remove Noise_P & Obtain noisy results

Figure 3.2: Query workflow of our system.

either ‘yes’ or ‘null’. For example, a query may specify that every client produces exactly 20 answers for the websites a user has visited in the last week, regardless of the actual number of visited websites. If a client has not visited 20 different websites, it generates ‘null’ answers for the difference. If it has visited more than 20 websites, then it cannot report on every website visited.

After collecting the encrypted answers from clients (step 3), the publisher generates DP noise separately for each bucket. Knowing these bucket definitions enables the publisher to generate the noise in the form of additional answers. It then mixes the real and noise answers (step 4), and forwards all answers to the data aggregator (step 5).

The data aggregator decrypts the answers, computes the histogram of bucket counts, and adds DP noise to each count (step 6). After signing the result, it transmits the counts to the publisher (step 7). The publisher then subtracts the noise it originally added (i.e., the number of additional answers for each bucket) to obtain its own final counts (step 8).

In the end, the publisher and the data aggregator *both* obtain aggregate results for the query. Because of the noise, neither of them obtains an exact result: the publisher’s result contains the noise the aggregator added, whereas the aggregator’s result contains the noise the publisher added.

If the publisher or the data aggregator wishes to release a result to the public, then they release the “double-noisy” result that was passed to the publisher in step 7. This precaution prevents the publisher and the aggregator from computing the noise-free result by subtracting their own noise, should the other publish its “single-noisy” result.

3.6.1 Audits

Clients occasionally *audit* publishers to detect if a publisher is dropping client answers (Figure 3.3). To audit a publisher, the client generates and encrypts a nonce (Step 2), and

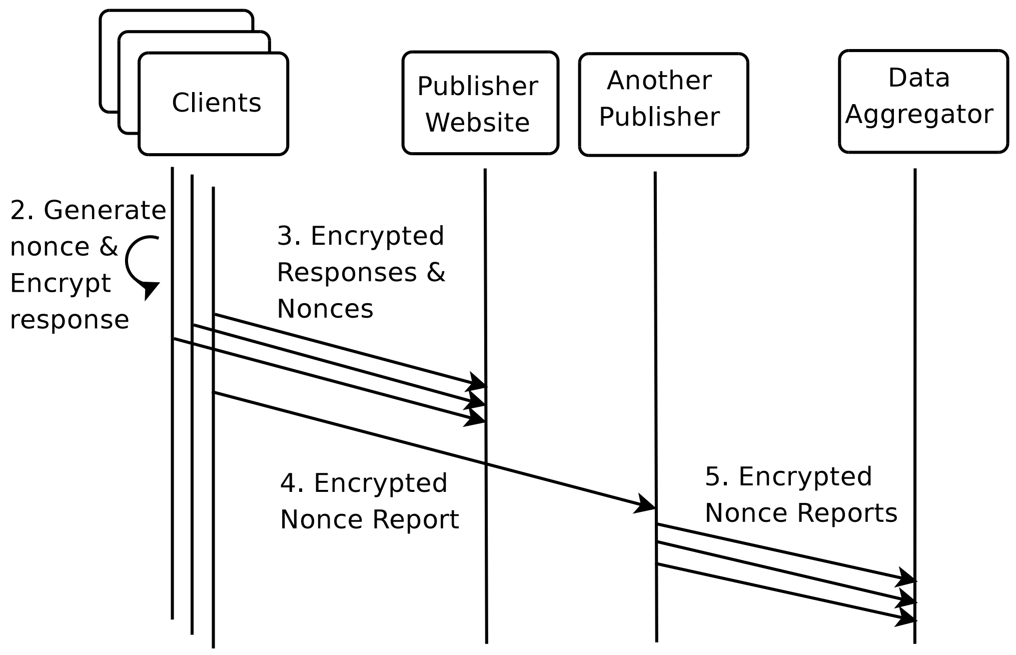

Clients
Publisher Website
Another Publisher
Data Aggregator
2. Generate nonce & Encrypt response
3. Encrypted Responses & Nonces
4. Encrypted Nonce Report
5. Encrypted Nonce Reports

Figure 3.3: Auditing mechanism of our system.

transmits it to the publisher instead of the answer the client otherwise would have sent (Step 3). The client also encrypts the nonce and the publisher to create a *nonce report*. This nonce report is then transmitted to another, randomly selected publisher (Step 4), which forwards it to the data aggregator (Step 5). If the data aggregator often receives nonce reports without the corresponding nonce answer, it suspects the publisher of dropping client answers.

3.7 Design

In this section, we describe how queries are generated and distributed, how the client generates a response and helps in auditing publishers, and how differentially-private noise is added by the publisher and the data aggregator.

3.7.1 Differential Privacy Background

A computation, C, provides (ϵ, δ)-differential privacy [104] if it satisfies the following inequality for all datasets D_1 and D_2 differing on one record and for all outputs $S \subseteq Range(C)$:

$$\Pr[C(D_1) \in S] \leq \exp(\epsilon) \times \Pr[C(D_2) \in S] + \delta \tag{3.1}$$

In other words, the probability that a computation C produces a given output is almost independent of the existence of any individual record in the dataset. In our setting, this dataset consists of the values of clients for a given attribute.

Differential privacy is achieved by adding noise to the output of the computation. This noise is independently generated for each component in the dataset. There are two privacy parameters: ϵ and δ. The trade-off between the accuracy of a computation and the strength of its privacy property is mainly controlled by ϵ: a smaller ϵ provides higher privacy, but lower accuracy.

The parameter δ relaxes the strict relative shift of probability. If $\delta = 0$, then the (ϵ, δ)-differential privacy falls back to the classical ϵ-differential privacy, which can be achieved by adding the Laplace distribution noise with a standard deviation $\sqrt{2}\Delta C/\epsilon$, where ΔC is the sensitivity of the computation, and is 1 for a computation counting set elements [103].

A non-zero δ is required in some scenarios where the inequality (3.1) cannot be satisfied [104]. Such (ϵ, δ)-differential privacy can be achieved in our system by adding the aforementioned Laplace distribution noise with a complementary resampling mechanism (Section 3.7.5.1).

Table 3.1: Query fields

QId	Query ID
p_s	Query selection probability
p_a	Audit probability
T_e	Query end time
$\mathcal{B}$	Set of answer values (buckets) each with ID b_i. ('null' and 'Not Applicable' (N/A) are well-known IDs)
A	Required number of answers
ϵ_P	differentially-private noise parameter for the publisher's result (used by the data aggregator)
ϵ_{DA}, δ	differentially-private noise parameters for the data aggregator's result (used by the publisher)
SQL	Database query

3.7.2 Queries

Publishers are required to list all their queries at a well-known URL on their website. This query list is signed by the data aggregator, even if there are no queries (i.e., an empty list). The aggregator may periodically check to ensure that the list is posted at the well-known URL (e.g., via fake clients it has deployed) to detect malicious publishers isolating clients by controlling the distribution of queries to clients. When a client visits a website, it retrieves the query list if the previous list has expired. Table 3.1 shows the fields contained in each query in the list.

QId is unique among all queries across all publishers working with this data aggregator. For each query in the list, the client decides whether to answer or ignore the query. This decision is made with the *selection probability* p_s assigned by the publisher, such that the publisher can obtain enough answers from its user base given the expected number of client visits. δ could be computed based on the expected number of answers. If the client decides to answer the query, then it separately decides whether to audit the query with *audit probability* p_a assigned by the data aggregator, by replacing the answer with a nonce. The *query end time* is the deadline for answering queries.

The buckets $\mathcal{B}$ are the potential answer values that are pre-defined. They may be downloaded separately and cached, if $|\mathcal{B}|$ is large. The aggregator sets the noise parameter ϵ_{DA}, and checks each query's ϵ_P to ensure it generates adequate noise before signing the query list.

The SQL query may produce zero or more numerical values or strings. Each bucket is defined as a numerical range or a string regular expression, such as salary ranges (numerical), or websites visited (string). A bucket is labeled as 'yes' if a row in the SQL output falls within the numerical range, or matches the regular expression. In addition, buckets have instructions to be followed when the same SQL output labels multiple buckets as 'yes' (e.g., select one or all), and the number of 'yes' labeled buckets exceeds the allowed number of answers A (e.g., select most frequently occurring or random buckets). If the client does not have A number of 'yes' labeled buckets, it uses a well-known bucket ID 'null'.

As an example, suppose a publisher wants to learn the age distribution of its female users. The SQL can be "`SELECT age FROM LOCAL_DB WHERE gender = female`". The buckets can be $\mathcal{B} = \{< 18, 18 - 34, 35 - 50, > 50\}$, and $A = 1$.

An SQL query often has predicates, such as "`WHERE gender = female`" in the above example. These predicates enable the publisher to query different segments of its user base. Too specific predicates, however, may produce results that are not useful in aggregate. To enable the publisher to notice that the predicates are too narrow, we define another well-known bucket ID 'Not Applicable' ('N/A'), used by the client when the query predicates fail. This well-known bucket ID is also useful for the data aggregator to detect malicious publishers who may set very specific predicates to isolate a client and repeat the query to overcome the noise (e.g., few answers with bucket IDs other than 'N/A').

3.7.3 Query Response

If the client software decides to answer a query according to a coin toss with bias p_s, it executes the SQL on its local database (step 2 in Figure 3.2) and produces the set of buckets labeled 'yes' (i.e., $\mathcal{M}$). If no predicates match, the client generates A answers with the well-known bucket ID 'N/A'. Each answer is individually encrypted with the public key of the data aggregator:

$$Response = Enc_{DA\_pub}\{QId, N/A\} \quad (A \quad times)$$

If the predicates match and $|\mathcal{M}| \leq A$, the client produces A individually encrypted answers, where $|\mathcal{M}|$ answers contain the matching bucket ID $b_i \in \mathcal{M}$, and $A - |\mathcal{M}|$ answers contain the well-known bucket ID 'null' :

$$Response = \begin{cases} Enc_{DA\_pub}\{QId, b_i\} & \forall b_i \in \mathcal{M} \\ Enc_{DA\_pub}\{QId, null\} & (A - |\mathcal{M}|\, times) \end{cases}$$

If $|\mathcal{M}| > A$, the client selects A buckets according to the instructions and produces A individually encrypted answers as its response.

To illustrate, assume a query asks the 20 most visited sites. If a client visited only 14 sites, it generates 14 answers with bucket IDs representing these sites and six answers with bucket ID 'null'. In contrast, if the client visited 25 sites, it generates only 20 answers for the 20 most visited sites.

Note that the combination of answer values (e.g., websites visited) may uniquely identify a client and allow the aggregator to track the user across different publishers asking this query. By individually encrypting each answer value, our system prevents the data aggregator from exploiting this information for tracking.

In certain cases, such combination of answers may be useful for web analytics. In these cases, the publisher and the aggregator can enumerate the combinations and use these combinations as answer values. Note that the number of clients with each unique combination may be low, such that the noise (Section 3.7.5) may dominate in the final result. As a result, the utility of the results may be reduced.

After generating the response, the client transmits it to the publisher along with the query ID (step 3 in Figure 3.2):

$$C \rightarrow P : QId, Response$$

The client then records that it answered the query so as not to answer it again before the query end time T_e. It also records the ϵ values to track the user's privacy exposure to the publisher and the aggregator. Note that the publisher may store client IP addresses answering this query to prevent a malicious client from skewing the aggregate result by sending many responses for the same query.

3.7.4 Audit Response

The audit serves two purposes. First, it can detect when a publisher is dropping client answers. Second, it can detect when a publisher is adding a substantial number of fake answers (beyond the noise).

The clients periodically generate nonces and obtain blind signatures from the data aggregator [95]. If a client decides to audit the publisher, it picks a nonce, encrypts the nonce and the QId with the aggregator's public key as well as $A-1$ 'null' answers, and transmits the response to the publisher as if it was a real response:

$$Response = \begin{cases} Enc_{DA\_pub}\{QId, nonce\} & once \\ Enc_{DA\_pub}\{QId, null\} & (A - 1 \quad times) \end{cases}$$

The client also randomly selects a different publisher, which is a customer of the aggregator. The client then transmits a separate, encrypted copy of the nonce and the nonce's blind signature *blind_sig* to that publisher:

$$NR = (Enc_{DA}\{QId, nonce\}, blind\_sig)$$

This nonce report cannot be directly submitted to the data aggregator, because the aggregator would learn which publisher a client has visited and decided to audit. Obtaining nonces in advance of the audit prevents the aggregator from correlating blind signature requests to nonce responses or reports. If the client has depleted its nonces with blind signatures, it generates a nonce, requests a new blind signature from the aggregator and delays its nonce response for a random amount of time until the query end time, T_e.

Each publisher periodically forwards received nonce reports to the aggregator. The client learns the set of other publishers by periodically downloading a list from the aggregator. This list associates a probability with each publisher that is roughly proportional to the number of answers each publisher handles. The client selects the different publisher according to this probability. As a result, each publisher handles a fair proportion of nonce reports.

If the aggregator consistently receives nonce reports via different publishers without a corresponding nonce message from the audited publisher, the aggregator suspects the audited publisher of dropping messages, possibly in an attempt to isolate a client.

In this case, the aggregator can validate this suspicion by masquerading as real clients from browsers it controls, and sending audits from these clients. This check is necessary because a malicious client may have sent nonce reports via different publishers, without the corresponding nonce via the audited publisher to cast suspicion on it.

The aggregator knows the probability of sending an audit response instead of a query response. As a result, it can estimate the number of clients answering this query by dividing the number of audit responses received by the audit probability p_a . It can also calculate the proportion of audit responses to query responses that should be received. If this proportion is consistently too low, then the aggregator suspects the publisher of adding additional fake answers.

The purpose of the blind signatures is to limit the rate a client can generate audits, which is helpful for two reasons. First, it drastically reduces the amount of suspicion a malicious client can cast on publishers by just sending the nonce reports, but not the nonces, as described above.

Second, by ensuring that these blind signatures are only assigned to clients and not publishers, the data aggregator prevents a malicious publisher from trivially generating many fake audits. Using these fake audits, the publisher could either drop client answers for an isolation attack without considering the possibility that they may be audit responses, or generate many fake answers by maintaining the right proportion of audits to answers. This use of blind signatures ultimately raises the bar for the publisher by forcing it to use botnet clients.

The blind signatures are timestamped to prevent an attacker from hoarding them for later use [75, 91, 110]. These timestamps are coarse-grained (e.g., end of the week) to prevent the aggregator from linking signatures to clients.

3.7.5 Noise Generation

3.7.5.1 Noise at the Publisher

The publisher generates differentially-private noise, rounded to the nearest integer, for *all* buckets using the data aggregator's noise parameters (i.e., ϵ_{DA}, δ), $\mathcal{N}_P = \{n_{P,1}, n_{P,2}, ..., n_{P,b}\}$, where b is the number of buckets (shown as Noise_P in step 4 in Figure 3.2). Remember that the publisher will forward the client answers to the data aggregator after adding noise. The client answers without noise would result in non-negative bucket counts when the aggregator decrypts and counts them. As a result, the mechanism for generating noise is to create additional answers. However, the amount of noise to add may be positive or negative.

If the noise is positive for a bucket, the publisher can generate that many additional answers with that bucket value, such that the data aggregator will obtain a count including the noise.

On the other hand, if the noise is negative for a bucket, the publisher cannot simply pick encrypted answers belonging to that bucket and drop them: the answers are encrypted with the aggregator's key, such that the publisher cannot know which bucket value an answer has. Furthermore, if there are no client answers with that bucket value, the count the aggregator should get should have a negative value (i.e., just the noise). This negative bucket count, however, cannot be achieved because there are no client answers with that bucket value for the publisher to drop.

To be able to generate also negative noise, we define an offset value o, which the aggregator will *subtract* from each per-bucket count. The number of additional answers supplied will be greater or less than this offset to create positive or negative noise, respectively.

To give an example, if the offset is 20, and the noise is +4, the publisher creates 24 answers for the given bucket, and the aggregator later subtracts 20 from the bucket's count. On the other hand, if the noise is -5, the publisher creates 15 answers. Stated precisely, the publisher calculates the number of per-bucket answers to create as:

$$\begin{aligned}\mathcal{N}'_{\mathcal{P}} &= \{n_{P,1} + o, n_{P,2} + o, ..., n_{P,b} + o\} \\ &= \{n'_{P,1}, n'_{P,2}, ..., n'_{P,b}\}\end{aligned}$$

These noise answers are encrypted with the aggregator's public key; hence, indistinguishable from client answers. After the query end time T_e, the combined set of client answers and noise answers $\mathcal{R}_{DA}$ are randomly mixed and sent to the aggregator along with the query ID and offset value:

$$P \rightarrow DA : QId, \mathcal{R}_{DA}, o$$

The aggregator decrypts the answers, counts them, and subtracts the offset to obtain the noisy result:

$$\begin{aligned}\mathcal{R}'_{DA} &= \{r_1 + n'_{P,1} - o, r_2 + n'_{P,2} - o, \\ &\quad ..., r_b + n'_{P,b} - o\} \\ &= \{r_1 + n_{P,1}, r_2 + n_{P,2}, ..., r_b + n_{P,b}\}\end{aligned}$$

where r_i is the count of client answers belonging to bucket b_i, and $n_{P,i}$ is the publisher's noise value for bucket b_i.

At this point, the data aggregator can make two checks to detect potential malicious publisher behavior. First, the aggregator can estimate the number of expected answers based on the number of audits received for this query and the audit probability p_a. After accounting for the expected noise answers (i.e., $b \times o$), if the received number of answers is significantly higher or lower than the expected number of answers, the aggregator suspects the publisher of adding or removing answers, respectively.

Second, after obtaining the bucket counts, the data aggregator can check for anomalies in this publisher's results. For instance, if the results for the same query consistently show low-value buckets along with high-value buckets (e.g., 'female<3', 'N/A>1K'), the publisher may be trying to isolate a client's answer and overcome the noise. In this case, the aggregator may suspect the publisher, check the query predicates manually, and/or may not return the result.

There remains the question of how to set the value of o. The noise value cannot exceed the offset, and must be resampled when $n_{P,i} < -o$. Even with this resampling, our procedure still provides (ϵ, δ)-differential privacy. The proof is elsewhere [81]. According to the theorem provided in [81], the offset o is set as:

$$o \geq \lambda \ln\left(\left(e^{\frac{A}{\lambda}} - 1 + \delta/(2A)\right) A/\delta\right) \tag{3.2}$$

where $\lambda \geq 2A/\epsilon_{DA}$. [125] argues that, for differential privacy guarantees to be met, $\delta < 1/c$, where c represents the number of clients answering this query. Since in our setting a client may answer the same query multiple times, we require $\delta < 1/(m \times c)$, where m represents the maximum number of times a client can answer the same query. For our purposes, we assume a conservative setting of $m = 1000$: if the same query is posed multiple times to adjust for changing user data and the query frequency is once per week, a value of 1000 for m corresponds roughly to 20 years. Recall that, as stated in our goals (Section 3.4.2), we do not aim to enforce a budget and provide strict differential privacy guarantees.

3.7.5.2 Noise at the Data Aggregator

After aggregating the answers and obtaining the noisy results, $\mathcal{R}'_{DA}$, the data aggregator generates Laplace noise using the ϵ value specified by the publisher (i.e., ϵ_P), $\mathcal{N}_{DA} = \{n_{DA,1}, n_{DA,2}, ..., n_{DA,b}\}$ (shown as Noise_DA in step 6 in Figure 3.2) for each bucket. The data aggregator then computes the double-noisy results $\mathcal{R}_P$:

$$\begin{aligned}\mathcal{R}_P &= \mathcal{R}'_{DA} + \{n_{DA,1}, n_{DA,2}, ..., n_{DA,b}\} \\ &= \{r_1 + n_{P,1} + n_{DA,1}, r_2 + n_{P,2} + n_{DA,2}, \\ &\quad ..., r_b + n_{P,b} + n_{DA,b}\}\end{aligned}$$

Then, this result is signed by the data aggregator and sent to the publisher (step 7 in Figure 3.2):

$$DA \rightarrow P : QId, \mathcal{R}_P$$

When the publisher gets $\mathcal{R}_P$, it removes its own noise and obtains its own noisy results, $\mathcal{R}'_P$, (step 8 in Figure 3.2):

$$\begin{aligned}\mathcal{R}'_P &= \mathcal{R}_P - \{n_{P,1}, n_{P,2}, ..., n_{P,b}\} \\ &= \{r_1 + n_{DA,1}, r_2 + n_{DA,2}, ..., r_b + n_{DA,b}\}\end{aligned}$$

where r_i is the count of client answers belonging to bucket b_i, and $n_{DA,i}$ is the aggregator's noise value for bucket b_i. In the end, the aggregator's result contains the differentially-private noise added by the publisher (i.e., R'_{DA}), whereas the publisher's result contains the differentially-private noise added by the aggregator (i.e., R'_P).

3.8 Analysis

3.8.1 Data Aggregator

Although the data aggregator follows the prescribed operation and does not collude with publishers, it may still be motivated to track clients across publishers and may try to exploit any information it learns. This information can include *identifiers* associated with clients, allowing the aggregator to track them. In the absence of a proxy, one such identifier is the client *IP address*. By using the publisher as an anonymizing proxy, our system hides IP addresses from the aggregator during the collection of answers.

The aggregator may try to obtain other identifiers by manipulating query parameters (i.e., $\epsilon_{DA}, \epsilon_P$, and p_a), and the audit activities (i.e., assignment of blind signatures and publisher probabilities for nonce reports). For example, an answer to a common query (e.g., a rare occupation) can be distinguishing among clients. The DP noise added by the publisher solves this problem (Section 3.7.5.1). To minimize this noise, the aggregator may set a large ϵ_{DA} value, but it would be easily detected by clients and industry regulators.

Besides rare answers, the *combination of answers* can also act as an identifier for a client. For example, a client's response to a query about most visited sites may be unique. Our system solves this problem by separately encrypting each answer at the client (Section 3.7.3) and mixing client answers with noise answers at the publisher (Section 3.7.5.1).

The aggregator has no incentive to use a large ϵ_P, which would only serve to reduce noise for the publisher.

In the auditing mechanism, the nature of the blind signatures and coarse-grained timestamps prevents the aggregator from connecting nonce reports back to clients. The aggregator may set a large audit probability for one publisher, and small probabilities for other publishers. This high probability would cause the clients of the first publisher to obtain blind signatures more often than others; hence, enabling the aggregator to infer the publisher these clients visit. However, unusually high audit probabilities will raise suspicion among clients and regulators. Furthermore, the utility of the first publisher will suffer, triggering suspicion.

3.8.2 Publisher

A potentially malicious publisher may want to exploit its position in the middle to learn an individual client's information, and falsify the results the data aggregator gets. The publisher can control the query parameters (i.e., SQL, A, $\mathcal{B}$, ϵ_P, p_s), the distribution of queries, the collection and forwarding of responses and nonce reports, the noise process, and the publishing of final results. We analyze how a publisher can try to exploit these parameters, and discuss how our system raises the bar for these attempts to succeed.

3.8.2.1 Publisher Attacking Clients

A client's response is encrypted with the aggregator's public key. The client also sends a fixed number of answers (i.e., A), preventing the publisher from learning how many buckets were matched for a query. Absent collusion, the publisher cannot learn an individual client's answer from the aggregator, and obtains only noisy aggregate results.

To minimize the noise the aggregator adds, the publisher may set a large ϵ_P value. By enforcing a maximum ϵ_P value, the aggregator can ensure that it will add enough noise to protect users' privacy (Section 3.7.5.2). Nevertheless, a publisher may try to learn a client's answer, by isolating it and repeating the same query to overcome the noise. We discuss how our system raises the bar for such a publisher.

Isolation via selectively dropping other clients' answers. To isolate a client's answer, a malicious publisher may drop answers from other clients and replace them with fake answers it generates. If the publisher drops some clients' answers, then it cannot be certain what the remaining answers are and cannot deduce what the isolated client's answer value is. If the publisher drops all the answers from all other clients, then these answers will contain nonces. When the aggregator consistently receives reports via other publishers, but not the nonces from the audited publisher, it suspects the publisher of dropping answers and can confirm this suspicion by masquerading as real clients and sending nonces through the publisher (Section 3.7.4).

To allow the malicious publisher to drop answers, other colluding publishers may drop nonce reports. However, they cannot selectively do so to help their partners, because they do not know about which publisher a given nonce report is. The aggregator also knows approximately how many reports a publisher should forward (i.e., via the publisher's probability to be randomly selected), and if it does not receive enough reports, it suspects the publisher of dropping them. For these reasons, a malicious publisher cannot easily help another malicious publisher to drop answers without detection.

Isolation via dropping target client's answer. A difficult, but theoretically possible attack is for the publisher to repeat a query and obtain results, half of which contain the target client's answer, and half of which do not. By comparing the average result of these two sets of queries, the publisher can determine if the target client's answer is positive or 'null'. The auditing mechanism may not detect this attack, because the audit is relatively rare, and thus, the target client may generate zero or very few audits. This attack is hard to carry out, because the client population may change over time, and because if the selection probabilty p_s is less than 1, different clients will answer different queries. In both of these cases, the non-noisy value would change a bit with successive queries; thus, requiring even more queries to eliminate the effect of noise.

Nevertheless, we simulated this attack, assuming a fixed set of 100 clients, one of whom is the target client. We execute the same query Q times, varying Q from 30 to 2000. We also vary the selection probability p_s to be 1.0, 0.5, and 0.05. When $p_s = 1.0$, we drop the target's answer half the time. For $p_s = 0.5$ and $p_s = 0.05$, there is no need to intentionally drop the target answer, because it is often naturally not provided. We average the counts for queries with and without the target. If the difference in the average is greater than 0.5, we guess that the target's answer is 1 (i.e., 'yes'). If it is less than 0.5, we guess that the target's answer is 'null'. We vary the number of queries Q, run 10000 trials for each Q, and calculate the percentage of times the guess is correct. This percentage is the publisher's confidence after Q queries.

Figure 3.4 shows the results of our simulation for the cases where the selection probability p_s is 1.0, 0.5, and 0.05 using $\epsilon = 0.5$. When $p_s = 1.0$, it takes over 350 queries to reach 95% confidence. Assuming one query per week, this attack would take roughly seven years. In the cases of $p_s = 0.5$ and $p_s = 0.05$, the attacker requires about 1000 and 2000 queries, respectively, for the same level of confidence.

Isolation via buckets or SQL. The publisher can also isolate a client by manipulating the query such that only the target client provides a positive answer (or, conversely, all clients except the target client provide positive answers). This attack can be accomplished either by manipulating the SQL predicate, or the bucket definitions (i.e., to include PII or a rare combination of attributes). Our general approach to both of these methods is to monitor answers for clues signaling that this attack may be happening, and to manually inspect SQL queries when these clues appear. While manual inspection is not ideal, we think that it will not be needed very frequently: Most publishers will probably ask the same types of queries. As a result, most of the queries will come from an already approved library, reducing the effort by the aggregator to monitor the queries and the buckets.

To start such a manual inspection when needed, the clue we are searching is any bucket whose count is consistently very low (roughly 0) or consistently very high

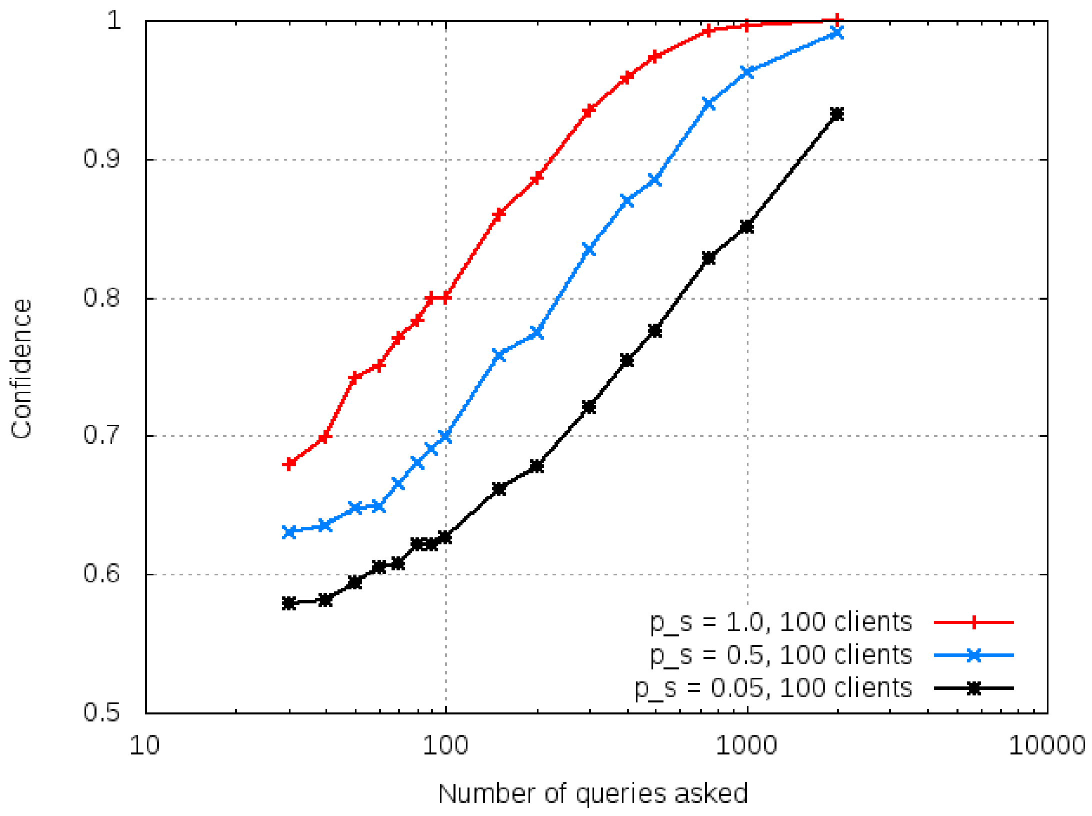


Figure 3.4: Confidence level for the isolation attack via dropping target's answer using the noise parameter $\epsilon = 0.5$.

(roughly the number of answering clients) for the same repeated query. For instance, if the predicate isolates the user (i.e., the user's name), then we expect to see very low bucket counts, except for the 'N/A' bucket, whose value will be very high. If the predicate does the reverse (i.e., includes all clients but the target), then the count of the 'N/A' bucket will be very low, and the other buckets will be very high. A very low 'N/A' bucket is suspicious, because in this case the predicate is apparently not needed and should be dropped. Likewise, if the target user is isolated by a rare bucket definition, then certain buckets will have very low counts. In this case, we can expect an honest publisher to modify its bucket definitions to prevent such consistently low counts.

Recall that a malicious publisher cannot just create an arbitrary number of fake answers to inflate the counts of these buckets: The aggregator would expect a certain number of nonces according to the audit probability of the query. To generate the right amount of nonce responses and reports, the publisher would require blind signatures (i.e., tokens) from the aggregator, who only assigns these tokens to clients and not the publishers. As a result, the publisher would be forced to use more clients (i.e., a botnet).

In some cases, however, examining the SQL may not be adequate. One such example is a predicate like "`WHERE page-visited = example.com/UniqueURL`", where `UniqueURL` is provided only to the isolated client. In this case, the aggregator must check that the URL is provided to multiple clients by operating fake clients. Nevertheless, such queries would generate high 'N/A' counts. Similar to the above case, we can expect an honest publisher to modify its bucket definitions to obtain more useful information and reduce such consistently high counts for the answer 'N/A'.

Isolation via query distribution. A malicious publisher may send a query to only one client. The aggregator, however, can ensure that the queries are available at well-known URLs at the publisher site via its own fake clients. Our auditing mechanism can also be extended to send reports when the query list or any queries are not accessible by clients.

Other attacks. By enforcing a maximum A value, the aggregator can ensure that clients do not spend unnecessary resources (e.g., CPU, bandwidth) while answering queries, to prevent denial of service attacks by the publisher.

There is no clear incentive for a selfishly malicious publisher not to add DP noise. Even so, the aggregator can still detect suspicious behavior: The aggregator can estimate the number of clients c_e as the number of audit reports received divided by the audit probability p_a. The approximate total number of expected answers is therefore $(c_e \times A) + (b \times o)$, where b is the number of buckets, o is the offset, and A is the number of answers per client. If answers are substantially lower than this value, the publisher is suspected.

A malicious publisher can publish its own single-noisy results that include only the aggregator's noise. However, these results will not have the aggregator's signature; thus, exposing the publisher. Furthermore, the aggregator can detect this behavior because it knows the double-noisy results.

3.8.2.2 Publisher Falsifying Results

To appear more popular or more attractive to advertisers, a publisher may want to falsify results by generating many fake answers. If the publisher exceeds the number of answers expected by the aggregator (i.e., $c_e \times A + b \times o$), it will be suspected. Thus, it can only generate answers that belong to certain buckets and is limited by the number of buckets and the offset (i.e., $b \times o$). This number may not be significant for queries with few buckets, depending on the total number of answers. For instance, it is 100 for a query about gender distribution with 5000 answers and an offset of 50.

On the other hand, $b \times o$ can be large for queries with many buckets. If all fake answers are used in few buckets, all other buckets would have values close to $-o$ after the offset subtraction. The probability of simultaneously generating these noise values is extremely low, signaling a manipulation. To prevent detection, the publisher would distribute the fake answers more evenly, limiting its distortion in a bucket.

3.8.3 Client

A client may act maliciously towards the publisher, the aggregator and to other honest clients. A client can lie in its response to distort the aggregate result; however, this distortion is limited by A set by the publisher. By keeping a record of client IP addresses, the publisher can also ensure that a client sends only one response for a query.

By sending fake nonce reports without the corresponding nonces, a malicious client can incriminate a publisher, and cause the aggregator to manually check this publisher. A client may also collude with a malicious publisher and generate nonce reports to help the publisher maintain the right proportion of audits to answers, either in an isolation attack, or in generation of fake answers to falsify results. By controlling the blind signature assignment to clients, the aggregator can limit this behavior, and force the publisher to get a bigger botnet, increasing chances of detection.

3.9 Implementation & Evaluation

3.9.1 Implementation

We implemented the **client** as a Firefox add-on. Our client keeps user information in a local database, looks for queries at a well-known URL (e.g., publishersite.com/queries/), and returns an encrypted response. The client is about 1000 lines of JavaScript code, excluding the 3000 lines of code for cryptography libraries for RSA.

The **publisher software** consists of a simple server-side script that stores the encrypted responses at the publisher's website, and a plugin for the opensource web analytics software Piwik [41]. Piwik already allows publishers to record various information about visitors, such as their browsers, operating systems and page views as well as the frequency of returning visitors. Our plugin extends Piwik's user interface to allow the publisher to view the queries, number of answers and results as well as enables the addition of the noise and forwarding of answers. In total, the publisher software is about 450 lines of PHP code.

The **data aggregator software** is a simple program that enables the publisher to upload the encrypted answers. The aggregator then decrypts and aggregates the answers, adds noise and returns the signed results to the publisher. Our implementation is about 275 lines of Java and PHP code.

3.9.2 Example Scenario

We analyze the computational and bandwidth overhead we impose on the components via some micro benchmarks. Lacking information about current aggregators' infrastructure makes a comparison difficult. Nevertheless, to analyze our system's overhead, we use the following scenario. Each week, a publisher poses queries shown in Table 3.2 to 50K clients. The first eight queries collect the same information current aggregators provide to publishers. The last three are *additional* queries our system enables the publisher to pose that are not available in today's systems: a 10-bucket histogram of the total number of pages visited by users across all sites, a 3-bucket histogram of visit frequency to each of 1000 websites selected by the publisher, and how many users use each of the top 5 search engines. We assume the aggregator uses a 2048-bit key.

Table 3.2: Queries and associated parameters. The buckets include our two well-known bucket IDs.

Property	# buckets	A	o
Age	7 + 2	1	66
Gender	2 + 2	1	66
Income	6 + 2	1	66
Education	5 + 2	1	66
Has children?	2 + 2	1	66
Location	5000 + 2	1	66
Ethnicity	5 + 2	1	66
Other sites visited	3000 + 2	10	751
Total # pages visited	10 + 2	1	66
Visit frequency	$(1000 \times 3) + 2$	10	751
Search engines used	5 + 2	3	211

Table 3.3: Per week bandwidth usage of the publisher and the data aggregator.

	Publisher	Data Aggregator
Collecting answers	0.37GB	-
Forwarding noise answers	1.20GB	1.20GB
Forwarding all answers	1.57GB	1.57GB

3.9.2.1 Computational Overhead

To measure the computational overhead, we ran our client on a laptop running Mac OS X 10.6.8 on an Intel Core 2 Duo 2.66 GHz as well as on a smartphone running Android 2.3.5 with a 1 GHz processor. Our JavaScript client can achieve about 380, 20, and 16 encryptions per second on Google Chrome, Firefox, and on the smartphone, respectively. Note that JavaScript can be slower than native code.

We ran the publisher and the aggregator software on a machine with 2GB of memory running Linux 2.6.38 kernel on an Intel Xeon two cores 2.4GHz. The publisher software can generate and encrypt around 7980 answers per second. In our scenario, the expected total number of additional answers (i.e., $b \times o$) for all 11 queries is around 4.9M, taking the publisher less than 11 minutes per week to generate.

The data aggregator software can decrypt and aggregate about 270 messages per second. In our scenario, the aggregation takes about 3.6 hours per week for the first 8

Table 3.4: Number of clients having used search engines on a given day in our deployment. Actual/Publisher/Data Aggregator

Day	Google	Yahoo	Bing	None/Other
01/18	72/73/73	20/20/19	1/-1/10	44/42/49
01/19	63/57/59	20/21/20	2/4/2	29/29/29
01/20	54/57/52	17/18/17	0/-1/-3	29/30/30
01/21	59/62/59	16/15/15	0/5/4	30/29/34

queries whose aggregate information current aggregators provide with tracking. The remaining three queries whose aggregate information is only available through our system take about 3.6 hours per week. Most of this overhead is due to the additional answers used as noise.

Note that all these numbers are obtained using a single CPU for the encryption and decryption operations. These operations can be easily executed in parallel.

3.9.2.2 Bandwidth Overhead

The compressed size of the biggest query (i.e., 5002 buckets) is about 35KB. In comparison, nytimes.com's homepage is about 500KB, excluding advertisements. Furthermore, buckets may not change very often, and can be cached.

The client's bandwidth overhead is in the order of a few kilobytes for sending responses. In our example, a client would consume about 8KB/week for all 11 queries. Table 3.3 shows the publisher's and the data aggregator's total bandwidth consumption per week. Most bandwidth consumption is related to the noise answers; however, the overhead is still acceptable: distributing nytimes.com's homepage to the 50K user sample *just once* would consume about 23.8GB whereas the collection of client answers and forwarding them with noise answers consumes about 1.57GB.

3.9.3 Deployment

To test our system's feasibility, we deployed our client via our friends and mturk.com for 15 days with 236 unique clients. We report on their browsing activities. On average, there were 118 active clients daily. Each day, we queried clients about how many pages they browsed, which sites they visited, their visit frequency to these sites, and which search engines they used. We used 3K most popular sites from Alexa and set $\epsilon_P = \epsilon_{DA} = 0.5$. Note that our goal was to gain experience rather than gather meaningful data.

The clients in our deployment were fairly active; almost half of them having visited at least 100 pages. Major sites, such as google.com, youtube.com and facebook.com, were (as expected) reported more than many other sites. We also gathered some data on the usage frequency of these sites. Many users have visited google.co.in much more frequently than facebook.com or youtube.com.

Table 3.4 shows the number of clients having used a search engine. One can see that the noisy counts of the users do not deviate much from the actual values, meaning that the relative error is not very significant (e.g., Google or Yahoo values). On the other hand, for low actual counts, the noise can dominate the results the aggregator and publisher obtain (e.g., Bing values). Our client covered only three search engines and might have missed searches on other sites with search functionality (e.g., Wikipedia).

Figure 3.5 shows the probability of generating a given noise value with the given epsilon values. We generated 1M noise values with each given ϵ value, counted the number of occurrences of each noise value and computed the probability of a noise value being generated. As one can see, the lower the ϵ value, the wider the range of the noise values. In other words, lower ϵ values increase the amount of noise added, which in turn gives more privacy.

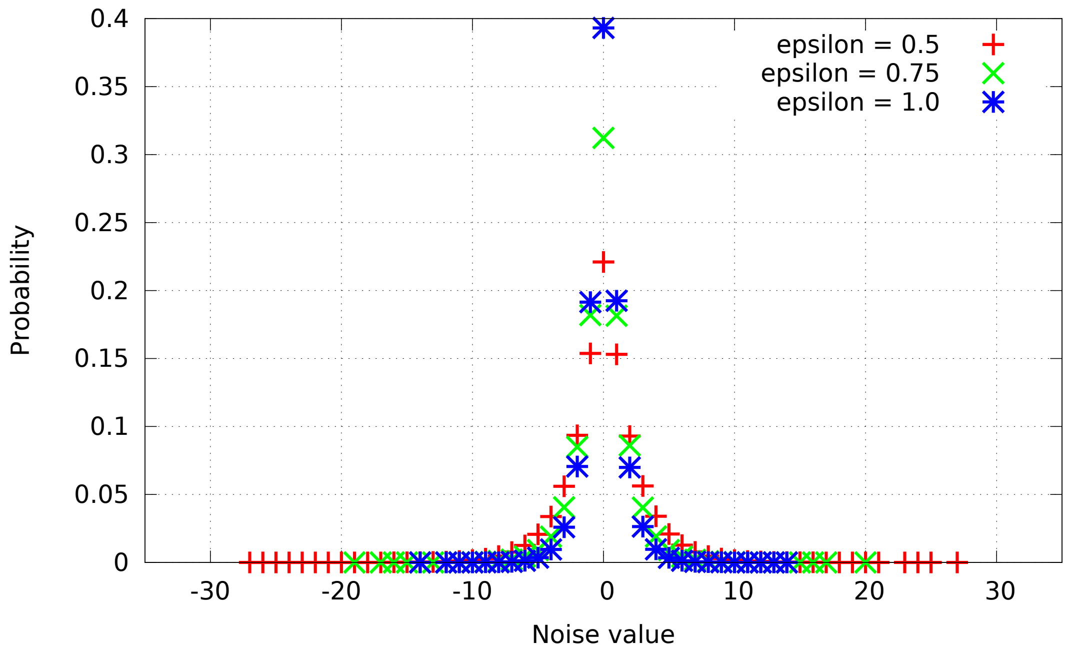


Figure 3.5: Probability density function of Laplace noise values with given ϵ values.

3.10 Conclusion & Future Work

We presented what is to our knowledge the first system for collecting accurate, extended web analytics without tracking users. Our system directly queries such data, while protecting user privacy by providing them with anonymity and unlinkability via the addition of Laplace noise. Our system utilizes the already-existing publisher as the anonymizing proxy, avoiding to require a new organizational component. It may be possible to apply our technique to other analytics problems, such as application analytics (e.g., mobile) and surveys about sensitive topics (e.g., elections, drug use). These scenarios, however, present additional constraints and challenges (e.g., developers without a website). We plan to examine them in more detail.

We envision that our system would be used in conjunction with first-party analytics software tools such as Piwik. These first-party tools can already provide the publishers with information about visitors on their sites (e.g., page views, browsers, operating systems, plugins, frequency of returning visitors). However, such systems do not provide important but potentially sensitive analytics information such as user demographics. Our system is designed to address this shortcoming by providing users with anonymity and unlinkability during the collection of this information. As a result, it complements the first-party analytics tools and provides the publishers with user demographics without having to track the users across the web.

While our design avoids the need for a new HbC organizational component (e.g., a proxy), it does so at the cost of certain new threats (e.g., publisher dropping responses) and additional mechanisms to make these threats more difficult. Even with an HbC proxy instead of a malicious publisher, however, the threat of isolation attacks through SQL or bucket manipulations remains. One avenue of future work is to explore new designs addressing these issues while maintaining the scalability properties of the current system, and to understand the trade-off points better.

One approach to mitigating the isolation attacks through SQL or buckets might be to simply withhold results for buckets with low values [83]. Another approach might be to have clients simply not answer repeat queries; however, this approach clearly results in a utility loss that needs to be better understood. Malicious publishers may also try to bypass such a mechanism via small variations in queries, essentially querying the same information with slightly different queries. Potential defenses may borrow ideas from information flow, each client tracking which piece of information it has exposed previously [87,133,145]. Other sophisticated approaches applied in centralized settings may help the aggregator and the publisher achieve better accuracy [85,118]. One avenue of future work is to understand whether we can extend their usability to our distributed setting.

An obvious limitation of our system is that the potential answer values (i.e., buckets) need to be enumerated and pre-defined before the publisher queries are distributed to clients, such that the clients can pick the most appropriate answer values after executing the queries. For queries about user demographics (e.g., age, gender, education level), this enumeration is not very difficult. On the other hand, for other queries (e.g., websites visited, search phrases, products viewed), it can become difficult. Other systems utilizing the same principle of 'queries with potential answer values' [97, 98] also suffer from this limitation. We address this limitation in the next chapter.

Chapter 4

Privacy-preserving String Discovery

In this chapter, we propose a system that enables the publishers to discover previously unknown string values that are present in client databases, but are difficult to enumerate. The system we presented in Chapter 3 as well as several others [97, 98] are effective in eliminating third-party tracking while still allowing publishers to query for extended analytics data. However, the queries need to have a list of pre-defined answer values, such that the client software can pick the most appropriate answer after executing a query. For many queries (e.g., visited websites, search phrases), this task of enumerating potential answer values can be difficult or impossible. The system we describe in this chapter addresses this shortcoming and complements the above systems. We describe our system's design, analyze its privacy properties and evaluate its feasibility using real-world data. A preliminary version of this work, including the design and evaluation, was published as a technical report at the Max Planck Institute for Software Systems [80]. The formal analysis of this work was also published as a separate technical report at the Max Planck Institute for Software Systems [79].

This chapter is organized as follows. The next section motivates the need for a privacy-preserving string discovery system for web and mobile analytics. Section 4.2

presents the challenges we face in designing such a system and our contributions to overcome these challenges. Section 4.3 introduces our definitions and the components in our system. In Section 4.4, we present our privacy and functionality goals. We list our assumptions in Section 4.5. An overview of our system and the building blocks we use are described in Sections 4.6 and 4.7, respectively. We present our system's design details, optimizations and duplicate detection mechanism in Sections 4.8, 4.9 and 4.10, respectively. A formal model and analysis of our system is given in Section 4.11. Section 4.12 describes our evaluation with real-world data. We conclude in Section 4.13.

4.1 Introduction

While statistics about user demographics (e.g., age, gender, income) are important, a new class of statistics is emerging: arbitrary text values or *strings*. Imagine a website publisher who wants to learn which (previously unknown) search phrases (e.g., 'pizza nearby') are used by how many of its visitors, or the developer of a photo application who wants to learn about the free-text tag values its users assign to their photos (e.g., 'dad and the cats'). Other examples include sites visited, installed applications and names of products viewed.

Some systems [83,90] try to tackle this problem via general-purpose secure multiparty computation (SMC) protocols [90], or expensive cryptographic operations [83], such as oblivious transfers (OT) [141]. Although eliminating potential privacy concerns about fully trusting a central entity such as a data aggregator, these operations put a significant load on the clients, whose resources may be limited in large-scale, distributed environments such as the web. In fact, users increasingly access the web via mobile devices with limited capabilities compared to personal computers [71–73]. Not supporting such clients will hinder the use of these systems for privacy-preserving analytics on the web and mobile settings. Furthermore, the main goal of the above systems is to aggregate and correlate network events among big organizations (e.g., ASes). This specialization limits the length of the strings these systems can handle due to the underlying cryptographic primitives. For instance, Sepia [90] and Applebaum et al. [83] assume a string length of 32 bits (i.e., the length of an IPv4 address). For longer strings as in our examples, these systems would require substantial changes.

4.1.1 Background: Privacy-preserving Analytics Systems

Our system described in Chapter 3 as well as other recent proposals for privacy-preserving analytics for web and mobile environments avoid the trade-off between privacy and scalability: With the help of a client software, they store user data at users' devices and release it in a protected fashion. These systems utilize less sophisticated but faster crypto operations than SMC or OT and can support a variety of client devices. Here, we describe the most relevant ones of these systems. We then explain their common limitation and how our privacy-preserving discovery system complements these systems.

***π*Box.** *π*Box [128] uses a trusted platform to restrict the interface for obtaining statistics from a mobile application: Application developers define a set of counter names. The platform enforces how much and how often a mobile application instance (i.e., client) can update these counters. The trusted platform also adds noise to counter values before reporting them to the developers. The system assumes that the counter names are well-known and pre-defined.

Hardt et al.'s system. Hardt et al. [117] propose a system to personalize mobile advertisements in a privacy-preserving way. To collect statistics, Hardt et al. use two honest-but-curious servers. The clients locally update a counter value for advertisement impressions (or clicks) and add noise to the values before sending the values for aggregation. The servers then aggregate the counter values. Again, the system assumes that the counter names are pre-defined.

PDDP. Chen et al. proposed a proxy-based system (PDDP) for querying user data. Similar to our system described in Chapter 3, user data is kept on user devices with the help of a client software. A separate entity, an honest-but-curious proxy, distributes queries from analysts to clients and collects responses that are encrypted by the aggregator's key. Chen et al. utilize a homomorphic encryption scheme, which allows the proxy to add noise to the responses blindly (i.e., it does not know how much noise is added). The queries, however, are distributed to clients with a list of potential answer values (i.e., buckets) like our system in Chapter 3.

SplitX. After our non-tracking web analytics system described in Chapter 3, a new system, SplitX, was developed in collaboration with others [97]; thus, its detailed architecture is not included in this thesis. SplitX utilizes the same idea of a client keeping user data on the user device and querying it as PDDP [98] and our non-tracking web analytics system. The biggest difference is that SplitX utilizes a more efficient encryption scheme instead of public key cryptography: XOR-encryption. After receiving the queries from the aggregator, the clients execute them over their local data and encrypt their answers with a fresh, randomly generated key. Both the XOR-encrypted answer and the key are then sent for aggregation to two non-colluding proxies, each proxy receiving one value (see details in Section 4.7.1). The proxies manipulate the XOR-encrypted answers to add differentially-private noise and forward the answers to the aggregator. The aggregator then simply decrypts and counts the answers.

The XOR-encryption improves the scalability of the entire system, including the aggregator. Although the addition of proxies may hinder the adoption of the system, it can also be considered a second improvement: besides contributing to the scalability, independently-run proxies also help the system to be more general. As a result, the system can be utilized not just for the web, but also for mobile applications. However, the queries still need to have a list of pre-defined answer values attached, like PDDP and our non-tracking web analytics system.

Common limitation. These systems are effective at eliminating third-party tracking while still providing useful statistics to analysts (i.e., web publishers, application developers). However, there is a common limitation in all of them: they require a list of pre-defined string values that are relevant to the user data in question. These string values represent the counter names in πBox [128] and in Hardt et al.'s system [117], such that the client software can update the correct counter value when necessary. In PDDP [98] and SplitX [97] as well as in our system described in the previous chapter, these string values correspond to the potential answer values (i.e., buckets) that accompany the queries when they are distributed to the clients, such that the clients can pick the most appropriate answer value after executing the queries.

Unfortunately, for many analytics scenarios (e.g., visited websites, search phrases, photo tags), this task of enumerating potential string values can be difficult or impossible. As a result, the applicability of these systems will be limited. The goal of the system presented in this chapter is to address this limitation and complement these systems. While we utilize a similar architecture like SplitX, our system does not require pre-defined string values for its operation.

4.2 Contributions

Our system described in Chapter 3 was designed specifically for web analytics purposes and used only existing entities. Other systems such as PDDP or SplitX, however, use independent proxies. These proxies make them more general, such that other analysts, such as mobile application developers, can also use the system.

In this chapter, we present the design and evaluation of a system that allows these analysts (e.g., web publishers, application developers) to discover previously unknown string values that are present in client databases, but are difficult to enumerate. The string values can be of arbitrary size without requiring any system or protocol changes.

Similar to our system in Chapter 3 as well as previous approaches [97, 98], the user data in our system resides at each user's own device running a client program. The client periodically participates in string discovery procedures by submitting its encrypted strings for aggregation. These strings, while still encrypted, are then counted by the entity providing the discovery service (i.e., the aggregator) and the two proxies. Strings with a (noisy) count above a discovery threshold t are then decrypted and provided to the analysts.

The aggregator and the proxies follow the protocol and do not collude with each other while running their operation. In this regard, they can be considered honest-but-curious. However, in our system, we allow these server components to run fake clients, because such actions may not be easily detected. We name this stronger adversary model as 'honest-but-curious with Sybils' (HbCwS). Our system has mechanisms to raise the bar for such adversaries and make their attempts at violating user privacy difficult.

The first challenge we face in designing such a system is to support a diverse set of client devices present in large-scale distributed environments like the web. To support even the client devices with limited computation and bandwidth resources, we employ a low-cost form of encryption (XOR), similar to SplitX [97]. Unlike SplitX, however, we do not rely on pre-defined string values.

To decide if a string value should be discovered, we need to count the number of clients with that value. The key challenge here is to count the clients without *revealing their strings*. In other words, we need to count the instances of a string value while the strings are still XOR-encrypted. To achieve this goal, we design a blind comparison method to distinguish encrypted strings, and count them without learning their values. We again avoid expensive operations at the clients and servers using low-cost XOR and hash operations.

Although this technique forces us to use pairwise comparisons resulting in $O(n^2)$ complexity, where n is the number of encrypted strings, our system utilizes two optimization heuristics to lower this cost in practice without sacrificing privacy. These optimizations take advantage of the assumed properties of the environment: the string distributions are likely to follow a power law (i.e., a couple of string values dominate n) and the number of possible string values is big.

Another challenge is to preserve the privacy of honest clients in the presence of Sybil clients. Such clients can be operated by an adversary, including the aggregator and the proxies, to artificially inflate string counts without being detected. As a result, low-cost options, such as clients sharing a secret with one server component to obfuscate their strings while another server component counts obfuscated strings [99], cannot be employed: the second component can learn the secret using fake clients and deduce the existence of clients with rare strings by pre-computing obfuscated values. Our design incorporates a *noisy threshold* technique to increase the difficulty of launching such attempts with Sybil clients on violating privacy.

We also need to prevent clients from manipulating encrypted string counts arbitrarily by sending the same string multiple times. Such malicious clients may want to reduce the utility of the results or act as Sybil clients to violate the privacy of honest clients. An environment such as the web contains millions of clients, which cannot be generally trusted to provide correct data and may have limited resources so that more sophisticated techniques such as zero-knowledge proofs [114] cannot be employed. As a result, effectively addressing the issue of manipulated counts becomes critical to the benefits of the analytics data. We describe the design of a duplicate detection mechanism that deals with this issue without increasing the computational and bandwidth load on honest clients.

To increase our confidence in our system's privacy properties, we formally model and analyze many aspects of our system using ProVerif [48]. In the situations where ProVerif cannot be used, we informally reason about our system's privacy properties. We demonstrate our system's feasibility using real-world datasets: website popularity from Quantcast [51] and search phrases from a large search engine. Our system causes several orders of magnitude less client computation overhead and reduces server computation overhead by at least two times compared to the closest system [83].

4.3 Definitions & Components

Before we describe our goals and assumptions, we define the following terms: A **string** is a text value present at the user's device. Some examples are 'google.com', 'pizza nearby' and 'spring in Paris'. A **string type** is the class of the string. The string 'google.com' may have type 'visited websites'. Similarly, 'pizza nearby' may be a 'search phrase' and 'spring in Paris' may be a 'photo tag'. A **generic string type** may be useful to many analysts (e.g., 'visited websites', 'search phrases'). An **analyst-specific string type** may be useful to one or a few analysts (e.g., 'photo tags in Instagram'). The **discovery threshold** t is the value that the noisy count of the number of clients with a given string must pass for the string value to be discovered.

There are three types of components in our system: client, aggregator, and proxies. Clients and the aggregator already exist in today's aggregation infrastructure. Proxies have been widely proposed for privacy purposes [83,97,98,116,117], and we also adopt this approach.

The client is a piece of software that stores user data (i.e., strings, string types) locally, similar to our system described in Chapter 3 and other systems [97, 98]. The client participates in discovery procedures by sending its encrypted strings. Note that the browser or mobile OS already sees user data.

The aggregator provides the string discovery service, which reports previously unknown strings and their noisy counts. Analysts may express their interest in learning strings of a string type. For example, an analyst may be interested in learning the search phrases users are using or websites users are visiting. These string values can then be used by the analysts to query distributed user data with other systems [97, 98] as well as with our system described in Chapter 3. The aggregator handles all interactions with the analysts, and controls access to the discovered strings (e.g., shares strings of an analyst-specific type only with that analyst).

The proxies provide clients with network anonymity, and enable the aggregation of encrypted user data and discovery of strings. They also help the aggregator limit the effect of malicious clients can have on string counts.

4.4 Goals

4.4.1 Privacy Goals

Our main privacy goal is to only learn string values that are reported by a sufficient number of clients, such that the count passes a threshold t supplied by the analyst while the aggregator enforces a minimum value for t. The **discovery threshold** t is defined as the value that the noisy count of the number of clients with a given string must pass, so that .

Our reasoning for this goal is two-fold: From a client's perspective, rare strings shared by few clients may leak privacy, and thus, should not be discovered. For instance, the tag 'Alec Finmeier getting drunk' is rarer than 'my birthday', and can leak a client's identity. From an analyst's perspective, the discovered strings may be more useful if shared by a relatively large client population. In our photo app example, the analyst (i.e., the developer) may only be interested in tags used by many clients. Thus, our goal is to *count* client strings *without revealing them*: any string value with fewer clients than a *discovery threshold* (t) should not be revealed to any component with high probability.

A fixed threshold, however, is not enough: to expose a rare string, an adversary (e.g., a component) can artificially inflate the count by creating t-1 Sybil clients. To prevent such attempts, our system should operate with noisy counts, and ensure that a string's noise-free count cannot be learned by a single component. In other words, the total noise value should be unknown to a single component.[1]

Additionally, the participation of the clients should be anonymous, such that given a string value or type, no component should be able to associate it with a client. It should also be unlinkable to prevent anonymous profiling of clients, such that given two string values or types, no component should be able to tell if they are from the same client. In addition, a discovered string should not reveal any information about any other string. For example, guessing a common string value should not leak any information about any other string.

Privacy Non-goals. Our non-tracking analytics system as well as many previous privacy-preserving analytics systems [97, 98, 117, 128] use differential privacy (DP) [102, 103] mechanisms to add noise to results. These systems provide users with some levels of formal DP guarantees. Unfortunately, using DP in environments like the web requires some relaxation for practicality [81, 97, 98]: there is no hard limit on how many times a client participates in the system (i.e., no budget). Like these systems, we do not enforce a budget, but use DP mechanisms (i.e., Laplace noise) to add noise to string counts. While

[1]Absent collusion among components (see Section 4.5 for details).

the clients in our system still record the ϵ values to track the theoretical privacy loss, we do not aim to provide users with DP guarantees. Although we wish to provide such formal guarantees, we think our goals and assumptions about the environment (i.e., the string distributions are likely to follow power law and the number of possible string values is big) align well for privacy-preserving discovery of unknown strings in practice.

4.4.2 Functionality Goals

Our main functionality goal is to help analysts by discovering unknown strings and reporting their noisy counts. Our system should scale well, both on the client and server sides. The client operations should not incur much overhead to support even the most resource-constrained devices (e.g., smartphones). To scale to potentially millions of clients with hundreds of millions of strings, server operations should also be fast. Finally, our system should limit a malicious client's effect on counts: a manipulated count can reduce the utility analysts obtain from the discovery and cause the discovery threshold to be ineffective.

4.5 Assumptions

In this section, we describe our assumptions for our system. Next, we describe similarities and differences between our system and SplitX [97], a high-performance private analytics system. Afterwards, we summarize our assumptions about the components in our system. These assumptions are not much different from SplitX. Finally, we list the assumptions about the string values that we want to discover.

4.5.1 SplitX

Our system is complementary to SplitX, but operates in a similar setting. In our design, we take advantage of the proxies that were introduced by SplitX and are required for its operation. We utilize the same XOR-encryption technique to support the client devices even with limited computation and bandwidth resources. However, unlike SplitX, our system does not rely on pre-defined string values for its operation and complements it by discovering unknown strings as potential answer values.

4.5.2 Client

The client typically runs on a user device, but may also run on another trusted platform. As we did previously and similar to previous systems [97, 98], we assume that the user trusts the client to protect the data it stores and regarding its operation, just as users trust their browsers for certificate handling and TLS connections.[2] A client can, however, be malicious and send the same string multiple times to try skewing its count.

4.5.3 Aggregator & Proxies

We assume that the proxies and the aggregator are honest-but-curious with Sybils (HbCwS): they follow the protocol, and do not collude with each other. However, each component may run fake clients to try to link/deanonymize other client strings.

Although our model is weaker than a more general model (i.e., arbitrarily malicious aggregator and proxies), we think that it reflects the reality on the Internet: The aggregator operates a business by providing string discovery service for analysts. The proxies can be operated by independent companies and/or privacy watchdogs. All of these entities would put their non-collusion statement in their privacy policies, making them legally

[2]We do not protect against malware infections on user devices.

liable. Moreover, any entity not following the protocol would risk losing reputation and customers. Previous systems make similar assumptions [81,83,97,98,101,116,117].

Finally, we assume the aggregator and proxies are not impersonated, and all end-to-end connections use TLS (i.e., no eavesdropping and no in-flight modifications).

4.5.4 String Values

We make the following two assumptions about the string values present at the client databases. First, we assume that the number of possible string values is big, such that an exhaustive enumeration of these values is very difficult. In fact, the main purpose of our discovery system is to handle analytics scenarios in which such an enumeration is difficult or impossible.

Second, we assume that the string distributions follow power law like many natural phenomena. As a result, we assume that a relatively small number of different string values will be present at a large portion of clients. As we show in Section 4.12.2, our real-world data about website popularity and search phrases support this assumption.

4.6 System Overview

Figure 4.1 shows an overview of our system. The aggregator periodically runs string discovery procedures. Clients periodically poll the aggregator with their string types (step 1 in Figure 4.1). These polls are XOR-encrypted and sent via the proxies to provide clients with anonymity and unlinkability: The aggregator cannot associate clients with string types, and cannot tell if any two requests are from the same client. Meanwhile, the encryption prevents proxies from learning clients' string types.

After receiving a poll request for a string type, the aggregator sends the associated

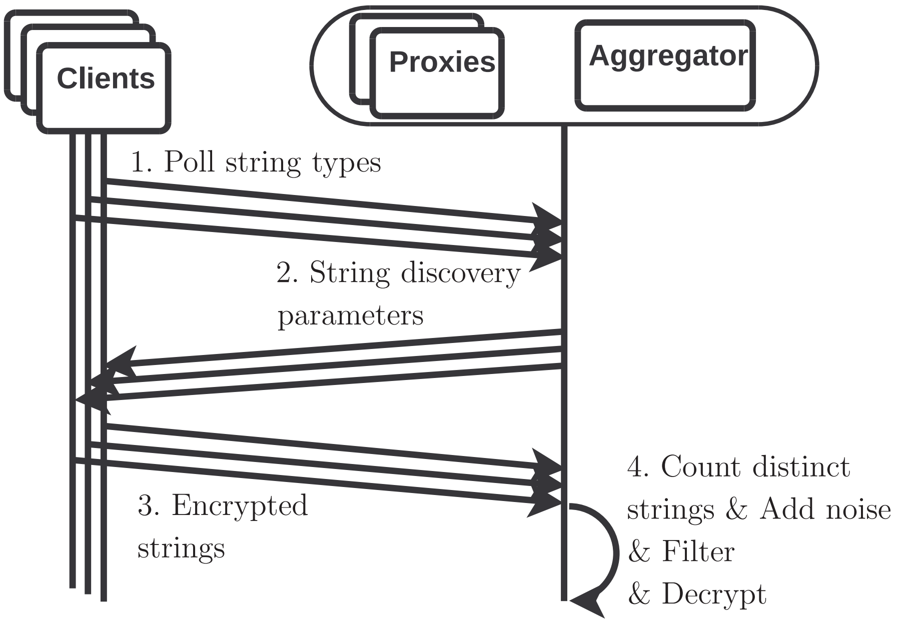

Clients
Proxies
Aggregator
1. Poll string types
2. String discovery parameters
3. Encrypted strings
4. Count distinct strings & Add noise & Filter & Decrypt

Figure 4.1: Overview of our system's operation.

(XOR-encrypted) string discovery parameters to the client via the proxies (step 2). The parameters include the ϵ value for Laplace noise and the discovery epoch that is used to synchronize the start and end times of discovery procedures for many string types. This synchronization serves as a checkpoint for the duplicate detection to limit malicious clients and helps the aggregator to group multiple string types together during aggregation, such that an adversary cannot deduce a client's string type just from the participation.

After getting the parameters, the client retrieves the strings belonging to the string type from its local database. The client then XOR-encrypts each distinct string with a separate, one-time key before sending it for aggregation (step 3).

During the aggregation step, our system utilizes a low-cost comparison method that only reveals if any two XOR-encrypted strings are equal. With this method, our system counts distinct strings, adds noise to their counts, and applies the discovery threshold—all without learning the actual string values. Strings whose noisy counts pass the threshold are then decrypted (step 4), and the aggregator reports them with their noisy counts to the analysts.

We employ two proxies, such that each proxy concurrently compares and counts the distinct strings of roughly half the clients, and independently adds noise to their counts. Consequently, the total noise added to a string count is unknown to any single component.

If implemented naïvely, the above protocol requires $O(n^2)$ comparisons to count n client strings. Each individual comparison is low-cost (i.e., XOR and hash), but the total can be prohibitive. We use two optimizations to lower this cost without violating our privacy goals in practice. Our optimizations exploit the properties of the assumed environment: First, the number of possible string values is big, such that an exhaustive enumeration of these values is very difficult. Second, the string distributions are likely to follow a power law like many natural phenomena.

With these optimizations, our system reduces the server computation overhead compared to the closest system [83], but increases bandwidth usage between the server components. Although the load for servers can be distributed, clients may be running on mobile, resource-constrained devices and become the bottleneck. With the increasing prevalence of these devices, supporting them becomes vital for scalability. We achieve this goal using low-cost primitives, which provide several orders of magnitude less computation overhead at the clients as well as support limited client bandwidth. By contrast, server bandwidth is less critical. For instance, data outgoing from EC2 is about \$0.09/GB up to 40TB, and even free when incoming [5]. Our system essentially trades off cheap server bandwidth for low client computation and bandwidth overhead.

A malicious client can try to exploit our system's anonymity and unlinkability properties to skew a string's count by sending it multiple times. Our system checks for such duplicates, potentially reported by any client, before counting strings. We utilize the

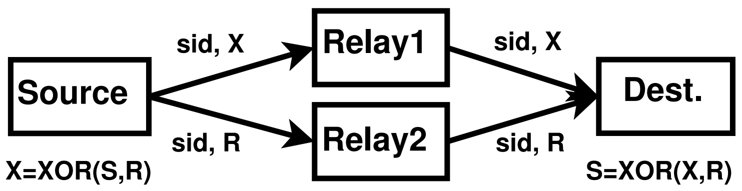


Figure 4.2: Splitting and joining. S is split to X and R. They are sent with the same *sid* via two relays.

same low-cost, blind comparison method mentioned above, and detect malicious clients without violating the privacy of honest clients.

4.7 Building Blocks

Here, we describe the low-cost XOR-encryption, our blind comparison method to determine the equality of two XOR-encrypted strings without revealing their values, and our noisy threshold mechanism to deal with Sybil clients. Section 4.8 presents our design details.

4.7.1 XOR-Encryption: Split & Join

Our system uses XOR as its crypto primitive like SplitX [97]. Splitting is equivalent to encryption, and joining is equivalent to decryption. These operations enable a source to anonymously send a string to a destination via two different, non-colluding relays. The relays do not learn the string value due to the encryption. Meanwhile, the crypto operations for the source and destination are low-cost (Figure 4.2).

To send a string S to a destination, the source splits S to obtain two split messages, X and R. Let L be the length of S. The source first generates a random, one-time key R of length L using a secure, one-time *seed* and a secure hash function H (e.g., SHA-2). Let h_i and $||$ denote the output of the hash operation at the ith iteration and the concatenation operator, respectively. The source starts hashing the seed and then applies the hash function to the output of the previous iteration until the desired length L is reached:

$$\begin{aligned}
h_1 &= H(seed) \\
h_2 &= H(h_1) \\
h_3 &= H(h_2) \\
&... \\
R &= h_1||h_2||h_3||...
\end{aligned}$$

The source then encrypts S with R:

$$X = S \oplus R$$

The source also generates a *split identifier sid*, a large random number (e.g., 128 bits) to ensure the two split messages will be uniquely paired by the destination with high probability. The source then sends X and R to the relays, who forward them to the destination:

$$\begin{aligned}
Source \rightarrow Relay_1 \rightarrow Destination &: sid, X \\
Source \rightarrow Relay_2 \rightarrow Destination &: sid, R
\end{aligned}$$

Borrowing notation from [97], we denote the split message pair $\{X, R\}$ as $\underline{S}$ (underlined S), and write:

$$Source \xrightarrow[Relay_2]{Relay_1} Destination : \underline{S}$$

The destination joins the split messages to obtain S:

$$S = X \oplus R$$

For efficiency, the source can send the $\langle seed, L\rangle$ tuple instead of R, and let the destination generate R.

4.7.2 Blind Comparison via pairwise-XOR and hash (PXH)

To count distinct string values without revealing them, our system uses a blind comparison method to determine the equality of any two XOR-encrypted strings. Consider two strings S_i and S_j with split message pairs $\{X_i, R_i\}$ and $\{X_j, R_j\}$, and split identifiers sid_i and sid_j, respectively. Recall that the split messages are held by two relays (i.e., X_i and X_j by Relay$_1$, and R_i and R_j by Relay$_2$). Let H be a secure hash function (e.g., SHA-2). For each relay, we define the pairwise-XOR hash (PXH) operation as:

$$\begin{aligned} PXH_{Relay_1}(sid_i, sid_j) &= H(X_i \oplus X_j) \\ PXH_{Relay_2}(sid_i, sid_j) &= H(R_i \oplus R_j) \end{aligned}$$

Recall that $X_i = S_i \oplus R_i$ and $X_j = S_j \oplus R_j$. Therefore:

$$\begin{aligned} PXH_{Relay_1}(sid_i, sid_j) &= H((S_i \oplus R_i) \oplus (S_j \oplus R_j)) \\ PXH_{Relay_2}(sid_i, sid_j) &= H(R_i \oplus R_j) \end{aligned}$$

If $S_i = S_j$, then $PXH_{Relay_1} = PXH_{Relay_2} = H(R_i \oplus R_j)$. By comparing PXH_{Relay_1} and PXH_{Relay_2}, our system can *blindly* determine if the original strings S_i and S_j are equal. Compared to just using the pairwise-XOR value, the secure hash ensures that one string cannot be reverse-engineered, even when strings are unequal and the other string is easily guessed (e.g., a common string).

Note that the X values of different strings must be held by the same relay (i.e., either Relay$_1$ or Relay$_2$). Otherwise, equal strings will not cancel out when the PXH operation is applied.

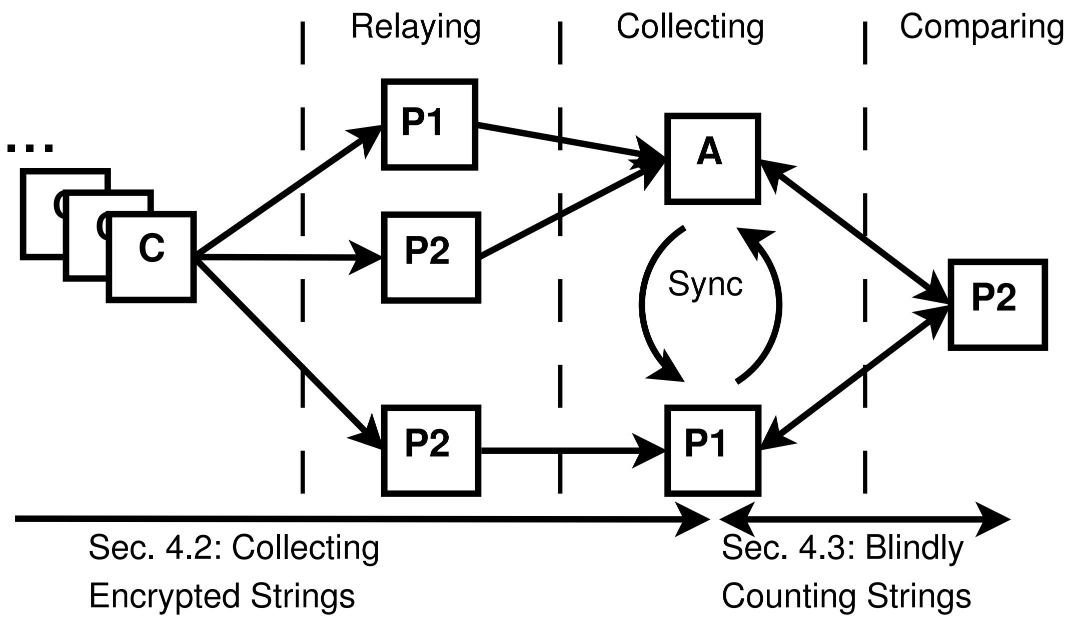
4.7.3 Noisy Threshold

Our system only decrypts strings whose noisy counts pass the discovery threshold t. We use Laplace noise, which is also used by differential privacy [102, 103, 105]. Adding Laplace noise to the output of a computation achieves the property that the probability of the computation producing a given output is almost independent of the existence of any individual record in the dataset the computation uses. In our setting, this property suggests the following: If there is a string with t-1 Sybils, the probability of the string being discovered (and decrypted) is almost independent of any honest client with that string value. In other words, whether a real client with that string value exists does not significantly affect its discovery.

4.8 Design

This section presents our protocol's details. First, the client receives the string discovery parameters (Section 4.8.1). Encrypted strings are then collected from the clients (Section 4.8.2). Afterwards, encrypted strings are blindly compared and counted (Section 4.8.3). Finally, noise is added to the counts (Section 4.8.4).

During these phases, there are three separate roles each component can perform: relaying, collecting, and comparing (separated by the vertical, dashed lines in Figure 4.3). The aggregator assumes only the collecting role, whereas the proxies assume all



Relaying
Collecting
Comparing
...
C
C
C
P1
P2
P2
A
Sync
P1
P2
Sec. 4.2: Collecting Encrypted Strings
Sec. 4.3: Blindly Counting Strings

Figure 4.3: Mirror operation 1 in our privacy-preserving string discovery system. The vertical, dashed lines separate the three roles. The arrows labeled with section numbers show the direction of information flow.

three roles (but not on the same data at the same time). Both proxies assume the relaying role between the clients and the aggregator.

To ensure no single component knows the total noise added to a string count, our system employs two proxies: each proxy compares and counts encrypted strings of about half the clients, and independently adds noise, which we refer as "mirror operations". For clarity, we present mirror operation 1 where Proxy$_2$ makes the comparison (Figure 4.3). We then describe how mirror operations enable us to *obliviously* add noise to counts (Section 4.8.4).

4.8.1 Initializing String Discovery

The client periodically polls the aggregator and receives *string discovery parameters* (SDP) for string types (ST) present in its local database (Figure 4.4). The polling mechanism

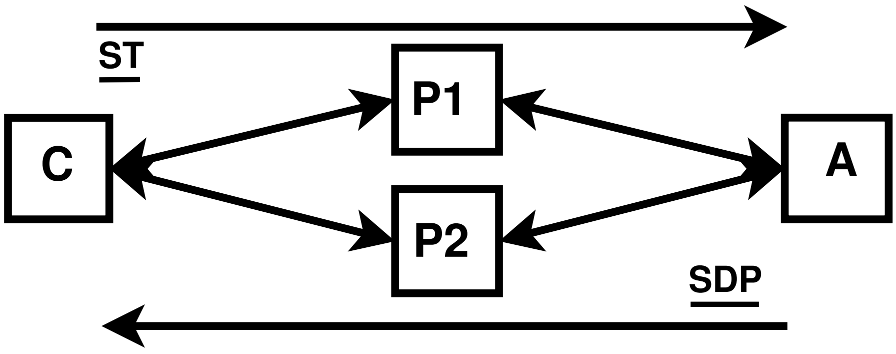


Figure 4.4: Initializing string discovery.

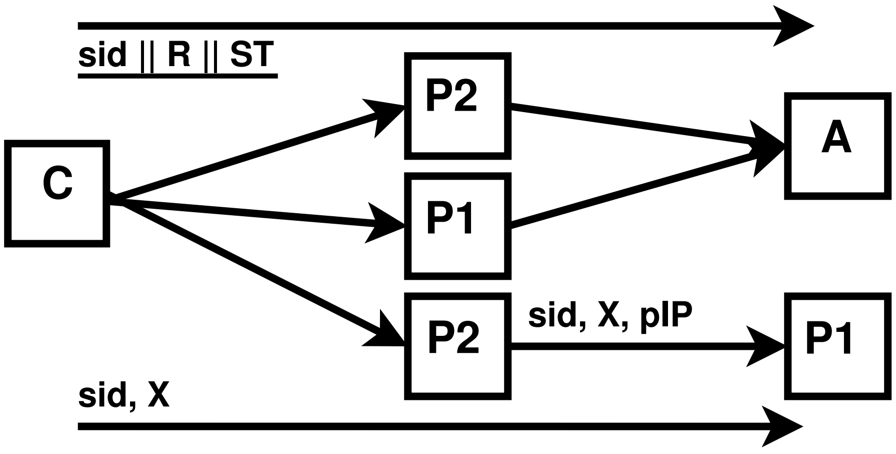


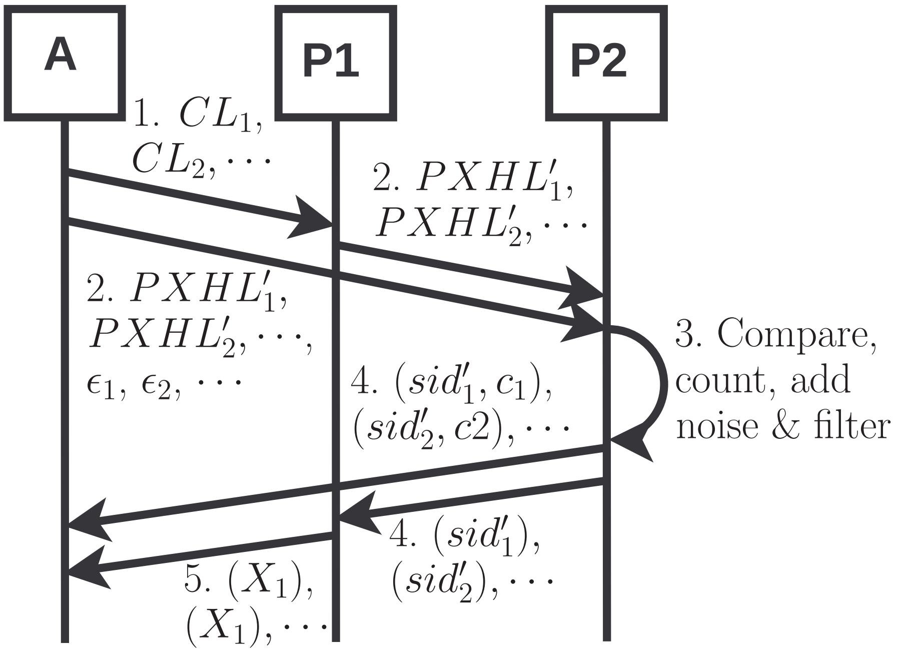
Figure 4.5: Collecting encrypted strings.

is similar to SplitX [97]. For each ST, the client creates a *separate* request, splits it, and sends it to the aggregator using the proxies as relays:

$$C \xrightarrow[P2]{P1} A : \underline{ST}$$

The aggregator splits the string discovery parameters associated with ST and sends them back via the proxies:

$$A \xrightarrow[P2]{P1} C : \underline{SDP}$$



A
P1
P2
1. CL_1, $CL_2, \cdots$
2. $PXHL'_1$, $PXHL'_2, \cdots$
2. $PXHL'_1$, $PXHL'_2, \cdots$, ϵ_1, ϵ_2, $\cdots$
3. Compare, count, add noise & filter
4. (sid'_1, c_1), $(sid'_2, c2), \cdots$
4. (sid'_1), $(sid'_2), \cdots$
5. (X_1), $(X_1), \cdots$

Figure 4.6: Counting and revealing strings.

The client joins the split messages to obtain the SDP, which contains ϵ and DT_{End}. ϵ is the privacy parameter to add noise to string counts. DT_{End} is the discovery end time (i.e., when no more strings are accepted). Discovery procedures are run in *epochs*: their start and end times are synchronized. A discovery spans only one epoch, but can be repeated. The aggregator can optionally add a list of hashes of previously discovered strings, such that clients only send undiscovered strings. If there is no discovery for a string type in the current epoch, SDP will be empty.

4.8.2 Collecting Encrypted Strings

To track the theoretical privacy loss (i.e., not enforcing the privacy budget), the client records the ϵ value from the aggregator. It then retrieves all strings of type ST from its local database. For each distinct string S, the client creates a split message pair $\{X, R\}$ and a split identifier *sid*. These values will be sent to the collecting components: the aggregator will receive *sid*, R and ST while Proxy$_1$ will receive *sid* and X. The aggregator will use the ST value to group encrypted client strings into comparison lists (Section 4.8.3). In an epoch, the client participates only in one randomly selected mirror operation (i.e., it uses the same collecting components).

Figure 4.5 shows the collection process. To prevent the aggregator from linking a client with a particular ST value, the client concatenates, splits, and sends the *sid*, R and ST values to the aggregator via both proxies:

$$C \xrightarrow[P_2]{P_1} A : \underline{sid \| R \| ST} \tag{4.1}$$

The aggregator joins the split messages to obtain the *sid*, R and ST. To anonymously send *sid* and X to Proxy$_1$, the client uses Proxy$_2$ as a relay: Proxy$_2$ assigns each client a temporary *pseudo IP address pIP* (i.e., valid only for the current epoch), and forwards *sid*, X and pIP to Proxy$_1$:

$$C \rightarrow P_2 : sid, X \tag{4.2}$$

$$P_2 \rightarrow P_1 : sid, X, pIP \tag{4.3}$$

The pIP values mark (encrypted) strings from the same client for duplicate detection (Section 4.10).

4.8.3 Blindly Comparing & Counting Strings

At this point, each collecting component in Figure 4.3 (Proxy$_1$ and the aggregator) has one split message of the XOR-encrypted strings and associated *sid* values. They exchange *sid* sets and discard unpaired split messages before proceeding with the blind comparison and counting.

Blind Comparison. As explained in Section 4.7.2, the blind comparison involves the computation and comparison of pairwise-XOR hash values (PXH_{P_1} and PXH_A) for each possible $\langle sid_i, sid_j \rangle$ tuple. The strings are not revealed during the comparison; however, if they are equal, knowledge about one string can be used to infer the other.

Neither collecting component is suited to make the blind comparison, because they are assumed to operate fake clients sending known strings. These strings can be identified (e.g., via their *sid* values), and the comparison result with an unknown string can be exploited. For this reason, the comparison is performed by Proxy$_2$.

Proxy$_1$ and the aggregator share a random, temporary secret R_s (i.e., valid for one epoch). They overwrite the *sid* values as $sid'_i = H(sid_i \| R_s)$, where H is a secure hash function (e.g., SHA-2),[3] and compute the *PXH* values as:

$$\begin{aligned} P_1 &: \quad PXH'_{P_1}(sid'_i, sid'_j) = H((X_i \oplus X_j) \oplus R_s) \\ A &: \quad PXH'_A(sid'_i, sid'_j) = H((R_i \oplus R_j) \oplus R_s) \end{aligned}$$

This modification of *PXH* values does not affect the comparison result, but ensures that Proxy$_2$ cannot reverse-engineer the *sid* values by using the fixed *PXH* value of any two known R or X values sent by its fake clients.

Blind Counting. Figure 4.6 shows the process to count the encrypted strings. The aggregator first groups the *sid′* values into *comparison list*s (CLs). A comparison list consists of either a generic string type (e.g., 'websites'), or multiple different analyst-specific string types (e.g., 'photo tags' and 'health app tags'). This mixing of multiple analyst-specific string types provides clients with additional privacy properties regarding their string

[3]Alternatively, they can agree on a shuffled mapping of *sid* values to *location pointers*, and use them (lp'_i instead of sid'_i).

types. Each list is then sent to Proxy$_1$ (step 1):

$$A \rightarrow P_1 \quad : \quad CL_1, \cdots, CL_m$$

As described above, Proxy$_1$ and the aggregator compute PXH' values for each possible $\langle sid'_i, sid'_j \rangle$ tuple in each comparison list CL_k, with $\langle sid'_i, sid'_j \rangle$ tuples as identifiers.[4] Let $PXHL'_{P_1,k}$ and $PXHL'_{A,k}$ represent the list of PXH'_{P_1} and PXH'_A values for CL_k computed at Proxy$_1$ and the aggregator, respectively. These lists are sent to Proxy$_2$. The aggregator also sends the ϵ_k values (step 2):

$$\begin{aligned} P_1 \rightarrow P_2 \quad &: \quad PXHL'_{P_1,1}, \cdots, PXHL'_{P_1,m} \\ A \rightarrow P_2 \quad &: \quad PXHL'_{A,1}, \cdots, PXHL'_{A,m}, \epsilon_1, \cdots, \epsilon_m \end{aligned}$$

For each $PXHL'$, Proxy$_2$ determines the equality of the encrypted strings for each tuple by comparing PXH'_{P_1} and PXH'_A values, and creates *equality list*s: if strings with sid'_i and sid'_j are equal, they are put in the same list. From each equality list EL_i, Proxy$_2$ randomly selects a sid' value as a *representative string*, and records it with the count of equal strings in the list.

Proxy$_2$ then adds Laplace noise to each count using the ϵ value of the corresponding $PXHL'$, and discards the representative sid' values whose noisy counts are below the discovery threshold (step 3). Let c_i be the noisy count of the representative sid'_i of EL_i. Proxy$_2$ sends each sid'_i and c_i to the aggregator, but only sid'_i to Proxy$_1$ (step 4):

$$\begin{aligned} P_2 \rightarrow A \quad &: \quad \{\langle sid'_1, c_1 \rangle, \cdots, \langle sid'_i, c_i \rangle, \cdots, \langle sid'_n, c_n \rangle\} \\ P_2 \rightarrow P_1 \quad &: \quad \{sid'_1, \cdots, sid'_i, \cdots, sid'_n\} \end{aligned}$$

[4]Or they can use the $\langle lp'_i, lp'_j \rangle$ tuples as identifiers.

Proxy$_1$ then sends the split messages (i.e., X values) of the corresponding strings to the aggregator (step 5):

$$P_1 \rightarrow A \quad : \quad \{sid'_1, X_1, \cdots, sid'_i, X_i, \cdots, sid'_n, X_n\}$$

The aggregator joins the locally held R_i and matching X_i for each sid'_i to obtain the discovered string values.

4.8.4 Mirror Operation & Oblivious Noise

We described the mirror operation 1 in Figure 4.7, in which Proxy$_2$ performs the comparison task. For roughly half the clients, Proxy$_1$ makes the comparison (i.e., mirror operation 2). Both proxies independently count, add noise to the count, filter strings lower than the discovery threshold t, and send them to the aggregator.

The mirror operations are mostly concurrent, except for two times requiring interaction: First, the duplicate detection (Section 4.10) requires synchronization between the proxies that relay X values (i.e., Proxy$_2$ in mirror operation 1 and Proxy$_1$ in mirror operation 2). Second, the comparing proxies send independently discovered strings to the aggregator at the end of the counting phase (Section 4.8.3).

It is possible that the aggregator receives a particular string value and its count only from one proxy (e.g., Proxy$_2$). If this count is published, Proxy$_2$ can associate the count with the string, and subtract the noise it added to get the string's noise-free count. If the aggregator adds its own noise and publishes the total, the total count can still indicate that the threshold was passed at only one proxy (i.e., if the total is less than twice the threshold). Proxy$_2$ could then use the ranking of the counts it reported, associate them with the strings (or eliminate most), and obtain the noise-free counts by removing its noise.

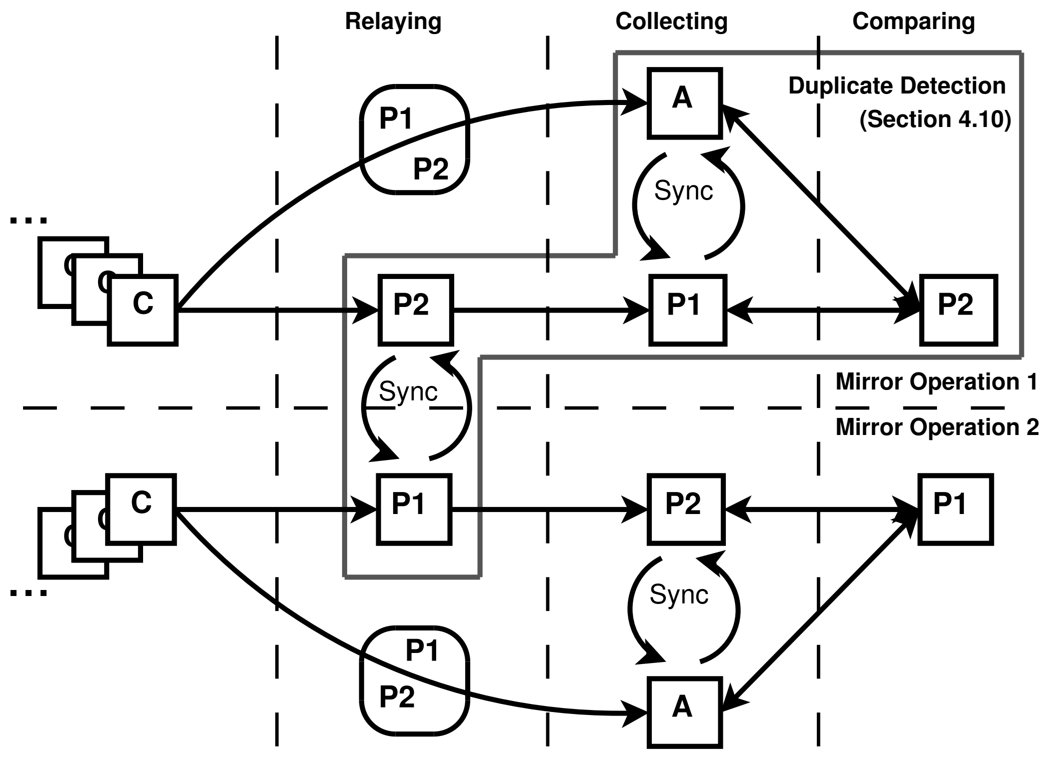


Figure 4.7: Complete system for privacy-preserving string discovery. The horizontal, dashed line separates the mirror operations. Components in duplicate detection are shown within the rectangular shape.

For this reason, the aggregator only publishes a string value if it is received from *both* proxies. That means, that the string value has passed the discovery threshold on both proxies. The aggregator then publishes the sum of both noisy counts (i.e., double-noisy count). The double-noisy count prevents the proxies from obtaining a string's noise-free count: even if a proxy somehow removes its own noise, the count will still contain the other proxy's noise.

Note that even the aggregator cannot learn a string value that did not pass the discovery threshold: if the aggregator received the string value, the string's noisy count must have passed the discovery threshold on at least one proxy.

4.8.5 Other Details

To prevent timing correlations, the proxies randomly order and delay split messages before relaying. The string length and the number of strings a client sends are selected from a well-known list (e.g., 50, 100) and can be parameters in the initialization based on the string type. Short strings are padded deterministically (e.g., with hash of string) before splitting. If a client has more strings, it randomly selects which strings to send. If not, it sends random *filler strings* with a modified string type (e.g., "tags_FS"), which are filtered by the aggregator. The client prepends the type to the string, distinguishing two analyst-specific strings even when the actual string values are the same.

4.9 Optimizations

Here, we present ways to reduce the total computation cost of comparisons without compromising our privacy goals in practice.

4.9.1 Sample-Identify-Count-Filter (SICF)

One heuristic is to use random samples to find strings with large counts and filter them. The high-level intuition is that, like many natural phenomena, the string distributions will show power law characteristics, and a few common strings will dominate in the comparison list. These strings can be identified with a small random sample, and strings equal to them can be filtered to shorten the list.

Figure 4.8 shows one iteration in mirror operation 1: The collecting components (Proxy_1 and the aggregator) first send the comparison list (CL') with modified *sid* values to the comparing component (Proxy_2) (step 1). Proxy_2 selects a random sample (S) (step 2), and sends it to Proxy_1 and the aggregator (step 3), who compute and send back PXH'

values for the strings in S (step 4). Proxy$_2$ then identifies the distinct (encrypted) strings in S, and selects one representative sid' value from each of the longest p equality lists in the sample (i.e., most common p distinct strings) (step 5). These sid' values are sent to Proxy$_1$ and the aggregator (step 6), who compute PXH' values for these p strings with all other strings in the CL' and send them to Proxy$_2$ (step 7). Proxy$_2$ counts and stores sid' values of all strings equal to each of these p strings (step 8), and sends the entire list of equal sid' values to Proxy$_1$ and the aggregator (step 9), who then filter them from the CL' (step 10).

This process can continue iteratively until 1) enough strings are discovered, or 2) Proxy$_2$ does not discover any new strings in step 8. When stopped, most common strings will have already been discovered. Some strings above the threshold may go undiscovered, but the probability of this event should decrease with bigger samples.

4.9.2 Short Hashes

Another heuristic is to distinguish strings before collection. The high-level idea is that the strings deemed different will not need to be pairwise compared. To achieve this task without compromising privacy, we let the clients map each of their strings into a *bucket* (B), using a hash function mapping to a *small* number of buckets (e.g., SHA-1 (mod 128)). The clients send each string's B value with its string type ST to the aggregator, who compiles the comparison lists with the distinct $\langle ST, B\rangle$ tuples: each list will be shorter, requiring fewer PXH operations in total.

The aggregator starts with one bucket and samples the encrypted strings. After the sampled strings are pairwise compared, the number of distinct strings will give the aggregator an idea on how many distinct strings to expect. The sample size can be increased for confidence. The aggregator then starts a new discovery with the decided number of buckets, and clients send their strings with their B values. Clients and

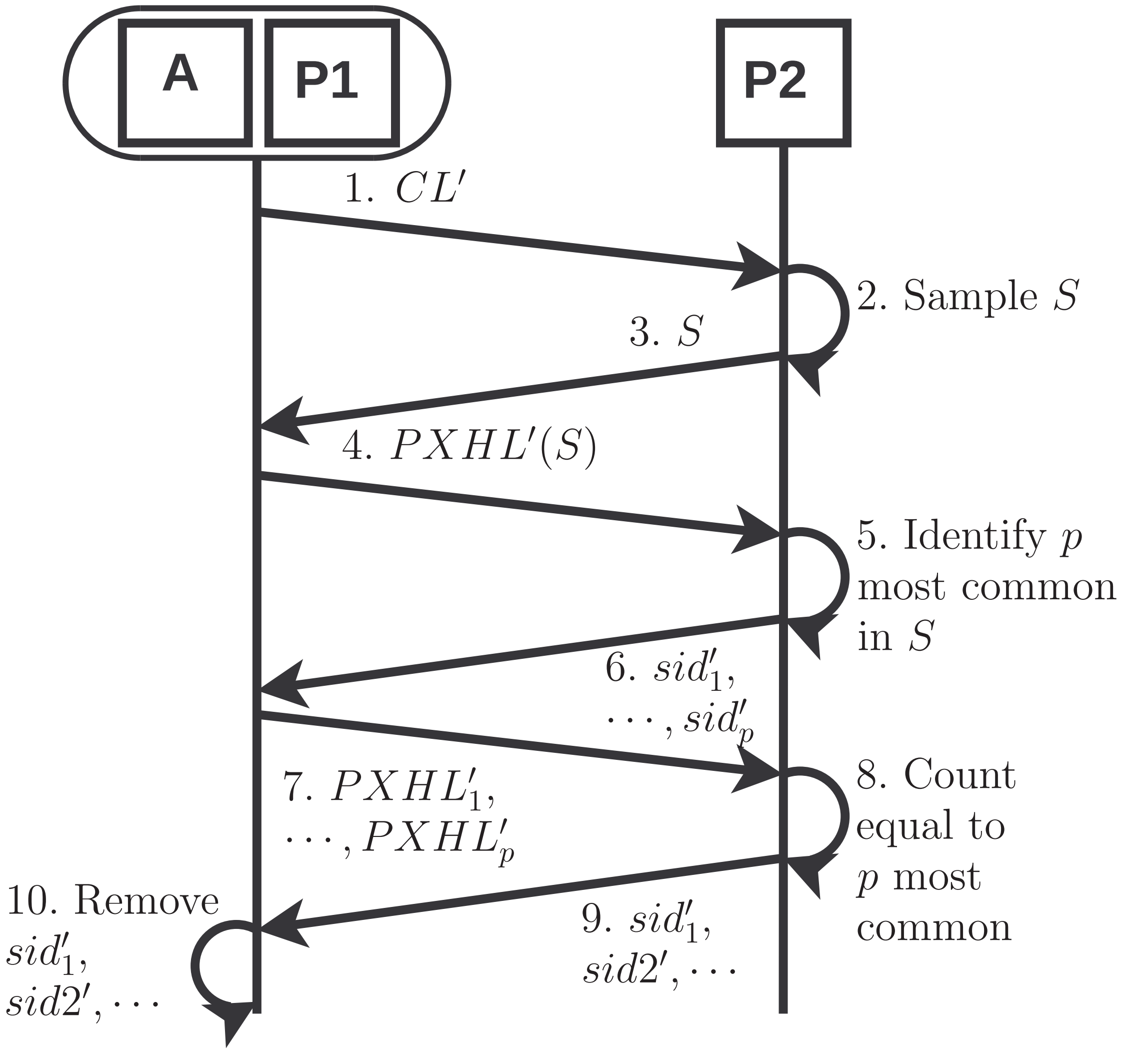

A
P1
P2
1. CL'
2. Sample S
3. S
4. $PXHL'(S)$
5. Identify p most common in S
6. $sid'_1, \cdots, sid'_p$
7. $PXHL'_1, \cdots, PXHL'_p$
8. Count equal to p most common
9. $sid'_1, sid2', \cdots$
10. Remove $sid'_1, sid2', \cdots$

Figure 4.8: Overview of our SICF heuristic.

watchdogs can set a maximum value for the number of buckets allowed (e.g., ≤128).

4.10 Detecting Duplicates

Our system detects malicious clients before counting the encrypted strings. Figure 4.7 shows a rectangular shape around the components involved in this phase. The high-level idea is to run the blind comparison protocol described in Section 4.8.3, but this time among all strings from a given client: equal strings will be duplicates, indicating a malicious client without revealing any strings.

We again describe mirror operation 1 for clarity. Recall that the client uses Proxy$_2$ as a relay for sending X values to Proxy$_1$ (Figure 4.5). Proxy$_2$ attaches a *pseudo IP address* (*pIP*) for each client IP address. Our protocol leverages these *pIP* values, and works in two stages.

Stage 1: The relaying Proxy$_2$ in mirror operation 1 and the relaying Proxy$_1$ in mirror operation 2 exchange real client IP addresses (left 'Sync' in Figure 4.7). If each client followed the protocol and participated in only one mirror operation in the current discovery epoch, the intersection of the lists will be empty. If not, the clients with IP addresses present in both lists might have sent a string multiple times. These clients' strings are invalidated by sending their *pIP* values to the respective collecting component (e.g., Proxy$_1$ in mirror operation 1), who then discards the associated X values.

Stage 2: After the invalidation, the collecting components (Proxy$_1$ and the aggregator) share another temporary, random secret R_{sdd} (upper-right 'Sync' in Figure 4.7). Using this secret, they overwrite the *sid* values (e.g., $sid'_i = H(sid_i \| R_{sdd})$).[5]

Proxy$_1$ then independently modifies the *pIP* values and gets a $pIP \leftrightarrow pIP'$ mapping to prevent Proxy$_2$ from linking the strings to the *pIP* values it assigned while relaying

[5]Or a different shuffled mapping for location pointers.

client strings. For each pIP', Proxy$_1$ sends the list of sid' values, $sidL'$, to Proxy$_2$:

$$P_1 \rightarrow P_2 \quad : \quad pIP'_1, sidL'_1, ..., pIP'_v, sidL'_v$$

The aggregator also independently modifies the actual string types to obtain an $ST \leftrightarrow ST'$ mapping. Multiple analyst-specific string types are mixed into one list for better privacy (i.e., multiple STs corresponding to the same ST'). Analyst-specific strings can still be compared safely, because the ST is prepended to the string (Section 4.8.5). For each ST', the aggregator sends the list of sid' values to Proxy$_2$:

$$A \rightarrow P_2 \quad : \quad ST'_1, sidL'_1, ..., ST'_t, sidL'_t$$

Using both $pIP' \rightarrow sidL'$ and $ST' \rightarrow sidL'$ mappings, Proxy$_2$ divides the sid' values into groups. For Proxy$_1$, each group $G_{P_1,i}$ corresponds to a unique $\langle pIP', ST' \rangle$ tuple. For the aggregator, each group $G_{A,i}$ corresponds to multiple (e.g., 20) pIP' values with the same ST'.

These groups are then sent to Proxy$_1$ and the aggregator:

$$\begin{aligned} P_2 \rightarrow P_1 \quad &: \quad G_{P_1,1}, G_{P_1,2}, ..., G_{P_1,n} \\ P_2 \rightarrow A \quad &: \quad G_{A,1}, G_{A,2}, ..., G_{A,n} \end{aligned}$$

Note that, even though Proxy$_2$ knows which unique tuples correspond to which groups, these tuple identifiers are not sent.

Proxy$_1$ and the aggregator compute the PXH' values using R_{sdd} for every possible $\langle sid'_i, sid'_j \rangle$ tuple in their respective groups, and send them to Proxy$_2$. Proxy$_2$ checks for equal strings belonging to the same pIP' value using the $pIP' \rightarrow sidL'$ lists. Recall that multiple pIP' values with the same ST' value are grouped together for the aggregator.

Using the $pIP' \rightarrow sidL'$ lists, Proxy$_2$ can identify the PXH' values that are redundantly computed by the aggregator and discard them, without affecting the duplicate detection. This redundancy ensures that the aggregator cannot anonymously profile a client with these string values, even if they are selected as representative strings by Proxy$_2$ at the end of the counting phase (Section 4.8.3), who cannot tell if they are from the same client or not.

Proxy$_2$ then sends the *sid'* values of these equal strings to the aggregator. In the counting phase, the aggregator independently modifies the PXH'_A values involving the duplicates with a random value rather than R_s, such that the blind comparison of the duplicates with other strings will yield 'not equal' and does not affect any counts.

NATs. Many clients may operate behind the same IP address (e.g., home/business gateways), making some duplicates legitimate. To decrease the bias caused by removing these duplicates, some randomly selected duplicates can be included based on the aggregator's policy.

4.11 Analysis

In this section, we present an analysis of our system. Our goal is to increase our confidence in our system's privacy properties. To this end, we formally modeled our protocol in applied-pi calculus [74] and verified our model using ProVerif [48]. We state our model's limitations and how these limitations might affect our verification results. For the parts we cannot model, we present an informal analysis and reason about why our system still achieves its privacy goals.

Table 4.1: Process grammar in applied pi calculus.

$P, Q :=$	processes
0	null process
$P \| Q$	parallel composition
$!P$	replication
new $n : t$; P	name restriction
if M then P else Q	conditional
let $x = M$ in P else Q	term evaluation
in(M,x:t); P	message input
out(M, N); P	message output
$R(M_1, ..., M_k)$	macro usage

4.11.1 Tools

Before we present the primitives used in our system and their equivalents in our formal model, we describe the formal tools we use to model our system.

4.11.1.1 Applied Pi Calculus

The pi calculus [134] is a language that is used to formally model distributed systems and reason about their interactions. The applied pi calculus [74] is an extension of pi calculus that is used to model and reason about cryptographic protocols. These distributed systems are modeled as a collection of parallel processes that exchange messages using channels.

Here, we describe some basics of the language (in conjunction with ProVerif) and how it is used to model interactions among concurrently running components of a system. The details of the language and how it is used in ProVerif can be found in the ProVerif manual [48].

Processes. A process is used to model the logical actions of a component in the system. The grammar to build processes is given in Table 4.1.

```
type hash.
fun H(bitstring): hash.
```

Figure 4.9: An example of a constructor without a destructor: one-way hash function.

```
type key.
fun senc(bitstring, key): bitstring.
reduc forall m: bitstring, k: key; sdec(senc(m,k),k) = m.
```

Figure 4.10: An example of a constructor with a destructor: symmetric encryption

Messages. Processes interact using messages. A message can be a name, a variable or the output of a constructor, or a combination (i.e., tuples). A name (e.g., 'string1') is used for atomic data. A variable (e.g., x) can be bound to a name or a message. Equivalence of two messages can be learned by applying an equation of the form $x = y$.

Constructors/Destructors. A constructor is a function that can be applied to names, variables and other messages. The corresponding destructor of a constructor ensures that a message can only be reversed into its original content (i.e., name, variable, message), if and only if the correct conditions are present. For example, one can model a secure, one-way hash function as a constructor without a destructor. Because there is no destructor, the output of this constructor cannot be reversed (Figure 4.9).

On the other hand, we can model the symmetric encryption *senc* with a destructor *sdec*, such that it will only output *m* when the key used to decrypt (i.e., *k*) is the same key used in the constructor *senc* (Figure 4.10).

Another way to model certain cryptographic primitives is to use equations. For example, one can model the symmetric encryption/decryption above as equations that capture the relationship between the constructors for all variables (Figure 4.11). Equations are less efficient than destructors, but are necessary to model certain cryptographic primitives that require algebraic relations between terms. One example is the Diffie-Hellman key agreement. Details of when to use constructors/destructors or equations can be found in the ProVerif manual [48].

```
type key.
fun senc(bitstring, key): bitstring.
fun sdec(bitstring, key): bitstring.
equation forall m: bitstring, k: key; sdec(senc(m, k), k) = m.
equation forall m: bitstring, k: key; senc(sdec(m, k), k) = m.
```

Figure 4.11: An example of a constructor with equations: symmetric encryption

Channels. Messages can be output and input on channels. Channels are asynchronized, such that the messages sent on a channel can be received out of order. A message m can be sent on a channel c using $out(c, m)$. Similarly, it can be input from the channel $in(c, r)$, such that the variable r will be bound to the message received from channel c. If the message m is a tuple of form (x, y), then the input action can be performed with a conditional, such that the variable y will be bound to variable z in $in(c, (= x_2, z))$ if and only if $x = x_2$.

ProVerif. ProVerif [48] is a tool for automated analysis of cryptographic protocols. Distributed systems modeled in applied pi calculus can be automatically analyzed to prove secrecy properties of these systems. ProVerif has been widely used in the literature to analyze properties of various cryptographic protocols (see the ProVerif manual [48] for a complete list).

ProVerif can perform reachability analysis of properties on an unbounded number of instances of the protocol. To do so, ProVerif overapproximates the state space of the protocol and explores it. As a result, when ProVerif claims that a property is true (i.e., no attack is possible), then it is true. In other words, ProVerif is sound. If ProVerif can prove a property is false, it generates an attack trace on why the property is not true.

ProVerif provides use of private channels. These channels are especially useful for modeling end-to-end encrypted channels between components, where the adversary is assumed not to have access to the messages.

ProVerif Limitations. Although ProVerif is sound, it is not complete. That means, ProVerif may not be able prove that a property holds. If ProVerif cannot prove that the property is neither true nor false, ProVerif states so.

Due to the overapproximation of the state space, it is possible that ProVerif finds an attack, although there is no attack possibility. In these cases, the attack trace can be investigated to confirm whether the attack is true or false.

As stated above, ProVerif performs the analysis with an unbounded number of sessions of the protocol. However, the repetition of actions cannot be supported, because repeated actions are translated into the same internal representation in ProVerif as non-repeated actions. As a result, there is no method for counting how many instances of the protocol ran until a point. This limitation prevents modeling of an adversary that might delay its attack until only after a certain number of messages have been received.

ProVerif cannot model traffic analysis. Additionally, privacy properties based on the 'hiding in the crowd' principle cannot be modeled. Although a piece of information may not be useful for a practical attack in a probabilistic sense, the mere fact that the adversary has access to it will trigger ProVerif to generate an attack trace.

4.11.2 Modeling Primitives

In this section, we describe the primitives we use throughout our model. We also state the limitations in the modeling of these primitives and how that might affect our verification results.

4.11.2.1 XOR-encryption

Our system uses XOR as its crypto primitive like SplitX [97]. Splitting is equivalent to encryption, and joining is equivalent to decryption. Recall that, in order to send a string S to a destination, the source splits S to obtain two split messages, X and R. Let L be

```
(* XOR-encryption and decryption *)
(* same as symmetric encryption/decryption *)
fun split(bitstring, bitstring): bitstring.
fun join(bitstring, bitstring): bitstring.
equation forall s: bitstring, r: bitstring; join(split(s, r), r) = s.
equation forall s: bitstring, r: bitstring; split(join(s, r), r) = s.
```

Figure 4.12: Formal definition of splitting and joining.

the length of S. The source first generates a one-time, random key R of length L and encrypts S with R:

$$X = S \oplus R$$

After receiving both split messages, the destination joins them to obtain S:

$$S = X \oplus R$$

We model our split and join operations simply as symmetric encryption and decryption (Figure 4.12).

Limitations of our XOR modeling. The exclusive-OR (XOR) operation is commutative and associative. In addition, equal strings cancel each other out when XORed together. These properties cannot be modeled in ProVerif explicitly. As a result, certain attacks making use of these properties cannot be explored by ProVerif.

Although the use of special channels (see Section 4.11.2.2 for details) allow us to work around the lack of cancellation property of XOR, it also weakens the ProVerif attacker: the attacker cannot arbitrarily XOR any two split messages it has access to and discover potential secrets.

We acknowledge these weaknesses in our model, but also point out that our protocol does not trust any one component to have both split messages before and during the

```
(* special type for the PXH result *)
type PXH.
(* comparison operations are irreversible because of the secure hash *)
(* therefore, they have no destructors *)
(* PXH operations for aggregator and proxy1 *)
fun computePXH(bitstring, bitstring, bitstring): PXH.
(* PXH operations with no secret *)
(* the following operation is only used to demonstrate the *)
(* 'known R values' attack by the adversary at proxy2 *)
fun computeKnownPXH (bitstring, bitstring): PXH.
```

Figure 4.13: Formal definition of the *PXH* operation. The third parameter is the secret shared between the collecting components (i.e., in our description, Proxy$_1$ and the aggregator).

discovery (i.e., X values are held by Proxy$_1$ and R values are held by the aggregator). As a result, one component cannot obtain the original string values or string types before the counts of the string values pass the noisy threshold. When there is no collusion among these components, which we assume, the adversary cannot access both values at the same time.

Furthermore, the R values are generated independently for each encrypted string: XORing just any two split messages will not yield anything meaningful. This case is similar to one component holding the random keys for symmetrically encrypted strings and the other holding the encrypted strings.

4.11.2.2 Pairwise-XOR and Hash (PXH)

To count distinct string values without revealing them, our system uses the pairwise-XOR and hash method to blindly determine the equality of any two XOR-encrypted strings. The pairwise-XOR and hash (PXH) operation works by comparing the $PXH_{Relay_1}(sid_i, sid_j)$ and $PXH_{Relay_2}(sid_i, sid_j)$ values for two strings, S_i and S_j, with sid_i

and sid_j, respectively. Recall that:

$$\begin{aligned} PXH_{Relay_1}(sid_i, sid_j) &= H(X_i \oplus X_j) \\ &= H((S_i \oplus R_i) \oplus (S_j \oplus R_j)) \\ PXH_{Relay_2}(sid_i, sid_j) &= H(R_i \oplus R_j) \\ &= H(R_i \oplus R_j) \end{aligned}$$

If $S_i = S_j$, then $PXH_{Relay_1} = PXH_{Relay_2} = H(R_i \oplus R_j)$, because equal strings will cancel out during the XOR operation (i.e., before the secure hash).[6] The formal definition of the *PXH* operation can be found in Figure 4.13.

While our *PXH* operation is straightforward, modeling it using ProVerif is not: Equal strings are supposed to cancel each other out; however, this functionality of XOR is not supported in ProVerif. Although there has been work on how to reduce protocols that use XOR semantics to a non-XOR version, such that ProVerif can be utilized [127], our approach is much simpler.

To overcome this lack of functionality, we utilize two special channels (Figure 4.14). We ensure that these channels are private (denoted as '[private]' after the declaration in Figure 4.14), and thus, not accessible to the adversary. These channels help us emulate the ideal functionality of the *PXH* operation, in which equal strings cancel each other out. These channels work as follows (Figure 4.15): Clients output their string values and split identifiers (i.e., *sid*) to a channel (i.e., *csid_str_map*). The aggregator outputs the *PXH* value of two encrypted strings along with the $\{sid, sid\}$ tuple to another channel (i.e., *cpxh_sid_sid_map*). The comparing proxy (in our model, Proxy$_2$), uses the *PXH* value to retrieve the original *sid* values of the strings from channel *cpxh_sid_sid_map*. It then uses the *sid* values to retrieve the original strings from *csid_str_map* and then compares

[6]In our description of mirror operation 1, these relays correspond to Proxy$_1$ and the aggregator. Here, we model the general *PXH* operation.

```
(* private channels used to determine equality of strings *)
(* no XOR support for equal strings canceling each other out *)
free csid_str_map: channel [private].
free cpxh_sid_sid_map: channel [private].
```

Figure 4.14: Special channels to emulate the *PXH* comparison due to incomplete XOR-functionality support in ProVerif. These channels are private (denoted as '[private]' after the declaration) and are inaccessible to the adversary.

```
(* client: process clientCollection *)
out(csid_str_map, (sidstr, str));
(* aggregator: process aggregatorComparison *)
out(cpxh_sid_sid_map, (pxhA, (sid1, sid2)));
(* comparing proxy (in this model, proxy2): process proxy2Counting *)
in(cpxh_sid_sid_map, (=pxhA, (sid1c: bitstring, sid2c: bitstring)));
in(csid_str_map, (=sid1c, str1c: bitstring));
in(csid_str_map, (=sid2c, str2c: bitstring));
```

Figure 4.15: Use of the special channels in each process.

their values. This way, we can emulate the cancellation property of XOR and determine whether any two strings are equal.[7].

Note that these channels are only used for the *PXH* comparison to determine the equality of encrypted strings. Even if we place the adversary at Proxy_2, our model ensures that the adversary does *not* have access to these channels or to the variable values obtained from these channels.

4.11.2.3 Noisy Threshold

Our model does not consider the threshold t and the noise that is added to the counts of encrypted strings. This noise is essential to ensure that an adversary cannot make the system reveal a string value whose count is artificially inflated (i.e., via fake clients) to be above the threshold. For example, the adversary may run $t-1$ fake clients and send the string value in a discovery procedure in an attempt to make its count go above the

[7]The other collecting component (i.e., Proxy_1) could also output the *PXH* values and the corresponding (*sid*, *sid*) tuples, but outputting these values once is sufficient because they are only used as lookup keys.

threshold to deduce the existence of a real client with that string value.

Unfortunately, we cannot use counts in ProVerif, such that we cannot model the threshold nor the noise. Our protocol depends on the Laplace noise that is used by differential privacy [102] to prevent this attack. According to Laplace noise as described in Section 4.7.3, the existence of a real client with a string value does not significantly affect the discovery of the string: The client may exist (i.e., the noise-free count is t) and the noise may be negative, and thus, the string may not be discovered. On the other hand, the client may not exist (i.e., the noise-free count is t-1) and the noise may be positive, and thus, the string may be discovered. These two cases are indistinguishable.

For a pattern to emerge, the discovery procedure needs to be repeated multiple times. The number of repetitions depends on the ϵ value, with lower ϵ values needing more repetitions. The aggregator also can ask the clients not send already discovered strings (Section 4.8.1), making the attack more difficult: the rare string from the real client will not be sent again after the first discovery, increasing the time required for this attack to succeed and making it impractical. High ϵ values for low noise can be detected by clients, watchdogs and proxies.

In our model, we abstract away the threshold and model the adversary's end goal of deducing the existence of a string by exploiting the comparison result. In our system, this goal is only achievable by Proxy$_2$, in which it may exploit the comparison result between an unknown string and a known string sent by one of its fake clients. Our ProVerif model then covers the cases, in which Proxy$_2$ can identify these strings and ensures that our protocol prevents this identification. We then reason about a case that cannot be modeled in ProVerif and reason why it is not a problem in practice according to our assumptions. As a result, the comparison result cannot be exploited by Proxy$_2$ (Section 4.11.3.5).

```
fun LINK(bitstring, bitstring): bool [private].
reduc forall a: bitstring, b: bitstring;
INFER_SYMMETRY(LINK(a, b)) = LINK(b, a).
reduc forall a: bitstring, b: bitstring, c: bitstring;
INFER_TRANSITIVITY(LINK(a,b), LINK(b,c)) = LINK(a,c).
```

Figure 4.16: Formal definition of linkability. The LINK function is private (denoted as '[private]' after the declaration) and is inaccessible to the adversary. After receiving the result of the explicit LINK function, the adversary can use the INFER_SYMMETRY and INFER_TRANSITIVITY functions to link variables that may not have been linked explicitly.

4.11.2.4 Datastores

We model a component's datastores as private channels. These datastores enable us to store the state of a component. We encode state information using messages and use the channel as a key-value store. Similar to Koi [116], our lookup operation consists of two parts. We first retrieve the message using the conditional lookup, which removes the message from the private channel (*cds*) and binds the variable *v* to the value of key *k* with the statement $in(cds, (= k, v));$. We then add the same message to the channel with the statement $out(cds, (k, v));$. As a result, we can keep the state information of a component in the channel, but can also perform operations according to the stored values.

4.11.2.5 Unlinkability

We model unlinkability in our system similar to Koi [116] (Figure 4.16). Any two variables that an adversarial component has access to at a single protocol step are explicitly linked using the private LINK function (denoted as '[private]' after the declaration in Figure 4.16). The result of this function is made available to the adversary via the *spyAtt* channel in the model (§4.11.3.1). We use the query functionality of ProVerif to see whether the adversary has access to the explicit linking information about two variables.

The adversary can also use two public functions to infer the linkability of two

```
fun EXISTS(bitstring): bool [private].
```

Figure 4.17: Formal definition of existence. This function is private (denoted as '[private]' after the declaration) and is inaccessible to the adversary.

variables. The INFER_SYMMETRY function models the symmetry property of the LINK function: if the variable a is linked to b, then b is also linked to a. The INFER_TRANSITIVITY function models the transitivity property of the LINK function: if a and b are linked to each other, and b and c are linked to each other, then a and c are also linked. Note that a and c may not have been accessible by the adversary at a single protocol step (e.g., collection of encrypted strings). The INFER_TRANSITIVITY function enables the adversary to infer linkability of such variables: a and b may be available at the collection of encrypted strings, and b and c may be available at the comparison and counting, and the adversary will still be able to link a and c together.

Alternative to this explicit LINK function, one can also make the LINK function public. This approach essentially would allow the adversary to link *any* two variables it has access to, without needing the INFER_SYMMETRY and INFER_TRANSITIVITY functions to achieve the same linking information. However, this approach leads to the adversary being able to link any two variables at *any* time of the protocol steps, leading to many false attacks. For example, assume the adversary is Proxy$_1$ and it runs its own clients. These clients' string types are naturally available to the adversary at Proxy$_1$. Proxy$_1$ also interfaces with honest clients, such that their network address is available. A public LINK function causes ProVerif to think that the adversary can link the honest client's address to the string type it knows from its own clients. As a result, many false attacks are reported by ProVerif.

4.11.2.6 Existence of a String

It is not possible to model counts in ProVerif. As a result, it is not straightforward to model an attack, in which the adversary creates enough Sybil clients and uses them to artificially inflate a string's count. This inflated count can then be used in an attack, such as deducing the existence of a string value at a real client. To overcome this limitation, we consider the end goal of the adversary, and model a specific function that denotes the existence of a string value (Figure 4.17).

This function is private (denoted as '[private]' after the declaration in Figure 4.17). In our model, whenever the adversary can deduce that a string value *str* exists (i.e., by knowing the comparison result is equal and knowing one of the compared strings), an EXISTS(*str*) message is emitted on the public channel the adversary has access to (i.e., *spyAtt*).

As a result, we can emulate the attacks where the adversary can exploit the comparison result and deduce that a string value exists at a real client.

4.11.3 Protocol Model & Verification Results

Here, we present the formal model of our system. For clarity, we describe the mirror operation 1, in which Proxy_2 is responsible for the comparison (Figure 4.3). We again consider our protocol without the optimizations. Later, we describe our model's limitations (due to limitations of ProVerif) and explain why our optimizations do not conflict with our privacy goals in practice.

In our model, we place the adversary separately at each component along our assumption that they are not going to collude. We then describe which variables are of interest to the adversary, which attacks are modeled, and any potential attacks ProVerif finds. We then reason about why some of these potential attacks are false attacks.

Table 4.2: Attacks found by ProVerif by various adversaries. The attack in the last row is known, but cannot be modeled in ProVerif.

Adversary	Attack	Found by ProVerif?	Validity (Reasoning)
Aggregator with fake clients	LINK(strtype1, strtype2)	Yes	False (Section 4.11.3.3)
	LINK(str1, str2)	Yes	False (Section 4.11.3.4)
Proxy$_1$	access to client IP	Yes	False (Section 4.11.3.3)
	LINK(pipC, ipC)	Yes	False (Section 4.11.3.4)
Proxy$_2$	access to client IP	Yes	False (Section 4.11.3.3)
Proxy$_2$ with fake clients	EXISTS(str1) (known *sid* values)	Yes	True with incomplete protocol (Section 4.11.3.5)
	EXISTS(str1) (known *PXH* values)	Yes	True with incomplete protocol (Section 4.11.3.5)
	EXISTS(str1) (count-as-a-signature)	No	True with very low probability (Section 4.11.3.5)

We divide our description along the lines of our protocol's phases. Each protocol phase builds on top of the previous one, such that the adversary can utilize the information it might have obtained during an earlier phase. For example, in the collection phase, the adversary still has access to the information that might have been exposed to the adversary during the initialization phase. All model files along with documentation can be found at the following address: `https://www.mpi-sws.org/~iakkus/private/verif/`

4.11.3.1 Adversary Model

The standard adversary in ProVerif has access to public channels and can observe only messages that are sent on those channels. In our system, components interact with each other using end-to-end encrypted channels that are modeled as private channels in ProVerif, preventing the adversary access to these variables. Other variables in compoents' internal states would not also be visible to the adversary. As a result, the standard adversary would not have access to any of the secret information in our protocol.

```
(* adversary can learn the client IP address? *)
(* (anonymity) *)
query attacker(ipC).

(* adversary can link the client IP to a string type? *)
(* (unlinkability) *)
query attacker(LINK(ipC, strtype1)).
query attacker(LINK(ipC, strtype2)).

(* adversary can link an anonymous client to two string types? *)
(* (anonymous profiling) *)
query attacker(LINK(strtype1, strtype2)).

(* adversary can link a client IP with a string? *)
(* (unlinkability) *)
query attacker(LINK(ipC, str1)).
query attacker(LINK(ipC, str2)).
query attacker(LINK(ipC, str3)).

(* adversary can link an anonymous client to two strings? *)
(* (anonymous profiling) *)
query attacker(LINK(str1, str2)).
query attacker(LINK(str1, str3)).

(* adversary can deduce that there exists a client with a string? *)
(* (existence of a string) *)
query attacker(EXISTS(str1)).

(* adversary can correlate a pseudo IP address to a real IP address? *)
(* (anonymity) *)
query attacker(LINK(pipC, ipC)).
```

Figure 4.18: Adversary queries in our ProVerif model. These items are queried throughout our model to see whether the adversary has access to the respective piece of information.

Our protocol, however, assumes that the components are honest-but-curious, meaning that they will try to learn as much information about clients as possible from the variables they obtain. As a result, we need a way to let the ProVerif adversary have access to the internal state of the component, where the adversary is modeled to be.

To do so, we use a public channel (i.e., *spyAtt*) similar to Koi [116]. Depending on the component we model as the adversary (e.g., adversarial aggregator), we emit messages that contain the internal state of that component on the public channel.

4.11.3.2 Adversary Goals

Figure 4.18 shows the list of queried items throughout our model. Depending on which entity is the adversary, some queries will be trivial cases leading to false attacks. For example, querying whether the adversary at a proxy has access to the client IP does not make sense: one of the tasks of the proxy is to provide clients with network anonymity. Table 4.2 gives a summary of the attacks ProVerif finds and cannot find as well as whether the attacks it finds are false along with the section numbers explaining the reasoning.

4.11.3.3 Discovery Initialization

Adversary at the Aggregator. To distribute the string discovery parameters to the clients, the aggregator needs to learn the string type. However, this information by itself is not useful to the aggregator, because it already organizes all discovery procedures and knows the string types from all analysts. Rather, the linkage between a client and its string types is of interest to the adversary. Additionally, the adversary at the aggregator may want to learn the string types a given client has, such that it can anonymously profile the client. At this phase, the clients do not send their strings yet.

ProVerif cannot find any attacks in the protocol as expected, because the proxies forward the client requests to the aggregator without exposing the client IP address.

The aggregator may run its own clients, whose string types it knows. When ProVerif is run, it finds an attack in which the aggregator can anonymously profile a client: the adversary can obtain the linkage between two string types (e.g., LINK(strtype1, strtype2)). When we investigate the attack trace generated by ProVerif, we find that the adversary accesses this information from the clients it runs. In other words, the aggregator anonymously profiles its own clients! When the adversary's clients are not used, ProVerif cannot find any other attacks.

Adversary at Proxy$_1$ or Proxy$_2$. The proxies provide the clients with network anonymity, and thus, see the client address. However, the requests containing the string types of the client are XOR-encrypted, such that the proxies do not see them. Generic string types (e.g., 'visited websites') are available at each client, such that the adversary does not gain any new information. For analyst-specific strings, the XOR-encryption prevents each proxy from learning the string type assuming there is no collusion between the proxies. Similarly, when the proxies forward the string discovery parameters back to the clients, they cannot obtain any information about the string types, because the parameters are also XOR-encrypted.

When the adversary runs its own clients, it naturally knows their string types. ProVerif finds the same false attacks as above. Besides these false attacks, ProVerif cannot find any other attacks.

4.11.3.4 Collection of Encrypted Strings

Adversary at the Aggregator. During the collection of encrypted strings, the aggregator receives the string type. This string type is used to compile the comparison lists. For example, multiple analyst-specific string types are put into the same comparison list. An adversary at the aggregator may want to obtain the client IP and link it to the string type it receives. However, the client sends its string type over the proxies in a split form. Therefore, it is similar to the discovery initialization and the aggregator cannot obtain the linking between the clients and their string types. At this point, the client strings are still XOR-encrypted, the aggregator holding the R value and the other collecting component (e.g., Proxy_1) holding the matching X value.

The aggregator may run its own clients, whose string types and string values it knows. Again, ProVerif finds an attack, in which the adversary can obtain the linkage between two string types similar to the initialization phase, but also with two strings (e.g., LINK($str1$, $str2$)). The investigation of the attack trace again shows that the adversary accesses this information from the clients it runs. Without the adversary's clients, ProVerif cannot find any other attack.

Adversary at Proxy_1. Similar to the above description, ProVerif finds false attacks about anonymous profiling when the adversary runs its own clients to participate in string discovery procedures.

Proxy_1 forwards the double-split messages that carry the string type to the aggregator, and therefore, interacts with the client directly. However, it does not see the client IP for the collection of the X values and only receives a pseudo IP pIP assigned by Proxy_2. ProVerif finds an attack in which the adversary can correlate the client IP with the pseudo IP address, because it has access to both variables. This attack is a false attack, because the system assumes that there will be many clients participating in a string discovery procedure. Any such linking will be *equally likely for any* client and thus, not meaningful. The probability of correlating the pIP value to the client IP will be inversely proportional to the number of cients.

The adversary sees the client IP address, because it is the proxy's task to provide the clients with network anonymity. Besides this trivial case, ProVerif cannot find any other attacks.

Adversary at Proxy$_2$. Proxy$_2$ assigns a pseudo IP address to each client and forwards the X values to Proxy$_1$. Besides the false attacks about anonymous profiling and a trivial case of Proxy$_2$ accessing the client IP address, ProVerif cannot find any other attacks.

4.11.3.5 Blind Comparison and Counting

Adversary at Proxy$_1$ or the Aggregator. The comparison and counting of the encrypted strings is done blindly using the comparison of PXH values. This comparison does not leak any information about the strings being compared, except for their equality or inequality. This comparison result is not learned by the collecting components (i.e., Proxy$_1$ and the aggregator), and they do not receive any new piece of information regarding string values and string types (besides the information they obtained in the previous stages of the protocol). ProVerif finds the same (false) attacks described above when the adversary is considered to be one of these components. Therefore, we do not discuss these cases explicitly.

Adversary at Proxy$_2$. On the other hand, Proxy$_2$ compares the PXH values and determines the equality of the strings. It puts equal strings into equality lists, such that if two strings are found to be equal, they are put into the same list. At the end of the counting process, Proxy$_2$ adds noise to each list's length length and filters the lists that are below the threshold. It then selects a representative string from each list and reports the count to the aggregator.

To perform these tasks, Proxy$_2$ learns the comparison result. Proxy$_2$'s role to make the comparison enables it to deduce the existence of a string (besides the other pieces of information it may want to learn) if the necessary conditions arise. In the rest of this section, we describe more specific attacks involving the adversary at Proxy$_2$. We introduce bugs to the original protocol and let ProVerif find the attacks, and explain how the original protocol prevents these attacks. We finally describe another attack, which ProVerif cannot find due to its limitation to model counts and how this attack is not of concern because of the system's underlying assumptions.

Known *sid* Values Attack. Proxy$_2$ may run clients and send known strings. It can determine that one of these strings is being compared with another string by utilizing the *sid* values the clients send and the collecting components (i.e., Proxy$_1$ and the aggregator) use as identifiers for the compared strings. We model this attack and verify that Proxy$_2$ indeed can use this approach to determine the existence of a string: ProVerif finds the attack and generates the attack trace.

Our original protocol prevents this attack by modifying the original *sid* values by overwriting them with a shared secret R_s between the collecting components, such that $sid'_i = H(sid_i \oplus R_s)$. When the original protocol is modeled, ProVerif cannot find any other attacks.

Known *R* Values Attack. Another method Proxy$_2$ can utilize to determine that it is comparing a known string value with another unknown string value is to use the R values.[8] When the *PXH* operation is applied without the secret shared between the collecting components, it is possible for Proxy$_2$ to identify $\mathrm{PXH}_{Aggregator}(sid'_i, sid'_j) = H(R_1 \oplus R_2)$. [9] Consequently, when a known *PXH* value is received by Proxy$_2$, it can deduce that sid'_i corresponds to sid_i and sid'_j corresponds to sid_j (or vice versa). Afterwards, when an unknown string is compared with one of these identified strings, Proxy$_2$ will be able to deduce the existence of a string.

Indeed, if the *PXH* values are computed without the secret between the collecting

[8]Or X values.
[9]Or $\mathrm{PXH}_{Proxy_1}(sid'_i, sid'_j) = H(X_1 \oplus X_2)$.

components, ProVerif finds the attack trace in which the adversary at Proxy_2 can launch this attack. The original protocol modifies the PXH values with the secret, such that $\text{PXH}_{Aggregator}(sid'_i, sid'_j) = H(R_1 \oplus R_2 \oplus R_s)$ and $\text{PXH}_{Proxy_1}(sid'_i, sid'_j) = H(X_1 \oplus X_2 \oplus R_s)$. With the original protocol, ProVerif cannot find any additional attacks.

Count-as-a-Signature Attack. Although we can model the above attacks and ProVerif is able to find them, there is another theoretical method for Proxy_2 to identify strings its clients sent: by using the count of strings as a signature. Proxy_2 can run clients to send a particular string. When the equality lists are formed, Proxy_2 can identify these strings from the list's length and identify their sid' values. Afterwards, Proxy_2 can utilize the comparison results of these strings with other unknown strings to deduce a string's existence.

Unfortunately, due to the lack of count support, this attack cannot be modeled in ProVerif. Although one can manually create a certain number of string instances, one cannot check the number of instances against a constant value.

This attack, however, is not a concern for the system in practice for the following reasons. The system assumes that the string distributions most probably follow a power law. This assumption means that there will be a long tail in the distribution, leading to a situation in which there are many strings with small counts. As a result, to create a unique signature for the string value injected via fake clients, Proxy_2 would have to create many clients. Additionally, the duplicate detection mechanism would force Proxy_2 to use one client for one string instance, increasing the difficulty of this attack even more. Finally, Proxy_2 would have to guess the number of actual clients with that string value that will choose Proxy_2 for the comparison, such that it can correctly estimate and identify the count as the signature. Even if it could do that, it still would not know how many clients with that string value have picked Proxy_1 for the comparison. We think that in practice, this attack will not be feasible with very high probability.

4.11.3.6 Duplicate Detection

To prevent a malicious client from arbitrarily manipulating string counts by sending the same string multiple times, our system performs a duplicate detection before it counts the encrypted distinct strings. The high-level idea is to run the same blind comparison protocol, but this time among all strings from a given client: equal strings will be duplicates, indicating a malicious client without revealing any strings. This phase is similar to the comparison and counting phase, in which Proxy_1 and the aggregator perform the *PXH* operations and Proxy_2 compares the *PXH* values.

Similar to the comparison and counting phase, ProVerif finds the same (false) attacks when the adversary is considered to be either the aggregator or Proxy_1. We do not discuss these cases any further.

As for the adversary at Proxy_2, it still learns the comparison result. However, this result is not exploitable by the adversary, because the comparison is performed only among the strings from the same client. That means, if Proxy_2 were to use fake clients and send strings, they would be compared with each other (and not with other honest clients' strings). As a result, Proxy_2 cannot exploit the comparison result to deduce the existence of a rare string value at an honest client.

Additionally, Proxy_1 and the aggregator use different secrets in the duplicate detection and comparison phases to modify the *sid* and *PXH* values. As a result, Proxy_2 cannot correlate the strings it compares in both of these phases.

4.11.4 Informal Analysis of the Remaining Protocol

Here we describe some parts of our protocol that were not modeled using ProVerif. We informally reason about why these parts do not affect our privacy goals in practice.

4.11.4.1 Mixing of Analyst-specific String Types for Comparison Lists

Our model does not consider the synchronization that takes place between the aggregator and the other collecting component (i.e., Proxy$_1$ in our description). During this synchronization, the aggregator mixes multiple analyst-specific string types into a single comparison list. While we could model this action in ProVerif, it would cause a number of false attacks due to the limitation of modeling 'hiding in the crowd' principle in ProVerif. Here, we reason about why this mixing helps us achieve our privacy goals in practice.

Recall that before the blind counting step, Proxy$_1$ receives each comparison list (CL) from the aggregator, such that it can compute the PXH values. However, it does not receive the associated string type ST for each list. Proxy$_1$ has access to the *sid* values used by its fake clients. It can use them to send strings with certain ST values and use these strings as a signal for the ST value of a CL. Any such information is uncertain, short-lived and anonymous: Multiple analyst-specific STs are mixed into the same CL, making Proxy$_1$'s guess uncertain (Section 4.8.3). Knowledge about a generic ST is not valuable, because every client has it. Proxy$_2$ anonymizes clients with a *temporary pIP* valid only for one epoch (Eqn. 4.3).

In ProVerif, the adversary at Proxy$_1$ would have access to its clients' string types. This knowledge would cause ProVerif to think that an honest client's string type is the same, just because they belong into the same comparison list. In a probabilistic sense, this deduction depends on the total number of string types being mixed into a single comparison list. Unfortunately, this probability cannot be modeled in ProVerif.

4.11.4.2 Mixing of String Types and PseudoIP Values in Duplicate Detection

Our system relies also on mixing the string types and pseudo IP values during the duplicate detection phase of our protocol. Similar to the previous case above, we have not modeled this part.

In Stage 2 of our duplicate detection mechanism, Proxy_1 sends the $pIP' \rightarrow sidL'$ to Proxy_2. The aggregator sends the $ST' \rightarrow sidL'$ mappings, where ST' corresponds to multiple analyst-specific string types. Proxy_2 in turn uses both mappings to send back groups of sid' values, such that a group for Proxy_1 corresponds to a $\langle ST', pIP' \rangle$ tuple and a group for the aggregator corresponds to multiple pIP' values with the same ST'.

During this procedure, Proxy_1 does not learn the ST' values in each group; only that they may be different. Furthermore, each ST' consists of multiple analyst-specific string types, preventing Proxy_1 from deducing the string type of a client. Similarly, the aggregator does not learn the pIP' values; a pIP' value may be in multiple groups and a group has multiple pIP' values.

4.11.4.3 Sample-Identify-Count-Filter Optimization

Our Sample-Identify-Filter-Count (SICF) optimization requires the model of counting support: the strings in each sample need to be counted, and the most popular strings are requested to compared with the rest of the strings to obtain a full count. For this reason, we did not consider this optimization in our model.

The reasoning about the privacy of this optimization is the following: To filter equal strings from the comparison list, Proxy_1 and the aggregator learn comparison results between some strings. If they identify one of these strings (e.g., their fake clients sent it), they can expose honest clients' strings that are equal to the identified string. However, they learn *many* sid' values of strings equal to *any* one of the p common strings, and thus, cannot be certain which strings are actually equal. In addition, these results belong to the *p* most common strings in the *random* sample, which reflects the string counts in the original comparison list. As a result, to expose a rare string, an adversary would need to send it so many times to make it one of the *p* most common strings in the sample. Our duplicate detection mechanism raises the bar for the adversary, forcing it to use more Sybils.

4.11.4.4 Short Hashes Optimization

It is straightforward to model our optimization that uses a small number of hash values. The hash value of the string only needs to be transmitted during the collection of the encrypted strings to the aggregator, similar to the string type. One can simply imagine that the string type already encodes the hash value. For example, one string type could be 'websites_with_hash_0', while another could be 'websites_with_hash_1', 'websites_with_hash_2' and so on.

The reasoning about the privacy of this optimization is the following: With a small number of hash buckets, many distinct string values will map to the same bucket (i.e., many hash collisions). Thus, the information gained about a string by knowing its bucket value will be small. For example, with 128 buckets, the average numbers of distinct string values per bucket in our datasets are about 7.8K for websites and 101K for search phrases. Clients and watchdogs can set a maximum value for the number of buckets allowed (e.g., ≤128).

4.12 Evaluation

In this section, we evaluate our system with real-world data. We first describe our datasets. We then evaluate our optimizations individually to show the benefits each optimization offers. We report on microbenchmark results, which we use to evaluate our system's overall feasibility. We compare our system with Applebaum et al.'s system [83] rather than [90] or [99], because it is the most similar system from previous work: it has centralized components that can run fake clients, offers some protection against malicious clients, and aims to provide aggregation in large-scale environments.

Applebaum et al.'s system has three components: client, proxy, and database. The client runs an encrypted, batched oblivious transfer (OT) protocol with the proxy to obtain encrypted and obliviously-blinded strings. During this process, these blinded strings are encrypted with the database's public key. The client then doubly-encrypts its strings as well as their values (i.e., '0' or '1'), first with the proxy's and then with the database's public key. It also generates a zero-knowledge proof (ZKP) [114] *per string* to prove that the values used were '0' or '1'. All ZKPs and encrypted strings are sent to the proxy, who forwards them to the database. The database verifies the proofs, decrypts the blinded strings and their values, and records them. If a blinded string's sum of values from all clients passes a threshold, the corresponding doubly-encrypted string is decrypted. Note that no noise is added to the counts of strings in this scheme.

4.12.1 Datasets

The strings in the first dataset are website names in a snapshot of Quantcast's top 1M sites in April 2013 [51], ranked by their visitor counts. The strings in the second dataset are about 13 million unique, anonymized search phrases from a large search engine[10], covering 3 months between 2011 and 2013. We assume each occurrence of a phrase is from a unique client. We label data based on the distributions of these datasets as "quantcast" and "search", respectively.

[10]We cannot disclose the name for confidentiality.

4.12.2 Benefits of Optimizations

4.12.2.1 Sample-Identify-Count-Filter

To gauge this heuristic's potential, we ran a test with 10M strings distributed based on our real-world datasets. The discovery threshold is 100, and the number of most common strings identified (p) is 20. We used samples with 99% confidence level and 3% margin of error, and stopped after 10 successive rounds of no newly discovered strings. This test validated our assumptions about string distributions and power law: 10% of the discoverable string values correspond to about 36% of all quantcast and 31% of all search strings. Our heuristic discovered most discoverable string values (97.5% and 93.3%) with effective speedups of 335.5 and 219.6.

4.12.2.2 Short Hashes

To show this heuristic's efficacy, we compute its speedup. The speedup S is the ratio of the number of *PXH* operations performed, without and with buckets. Let N be the total number of strings, n_i be the number of strings in bucket i, and B be the number of buckets.

$$S = \frac{\frac{N \times (N-1)}{2}}{\sum_{i=1}^{B} \frac{n_i \times (n_i - 1)}{2}}$$

Figure 4.19 shows the speedup increases with the number of hash buckets: Strings in separate buckets need not be pairwise compared, reducing the number of *PXH* operations. Due to the power law, some strings have large counts, increasing their respective buckets' counts. After 256 buckets, these buckets start dominating in the sum of *PXH* operations, reducing the speedup. By contrast, with a uniform distribution, the

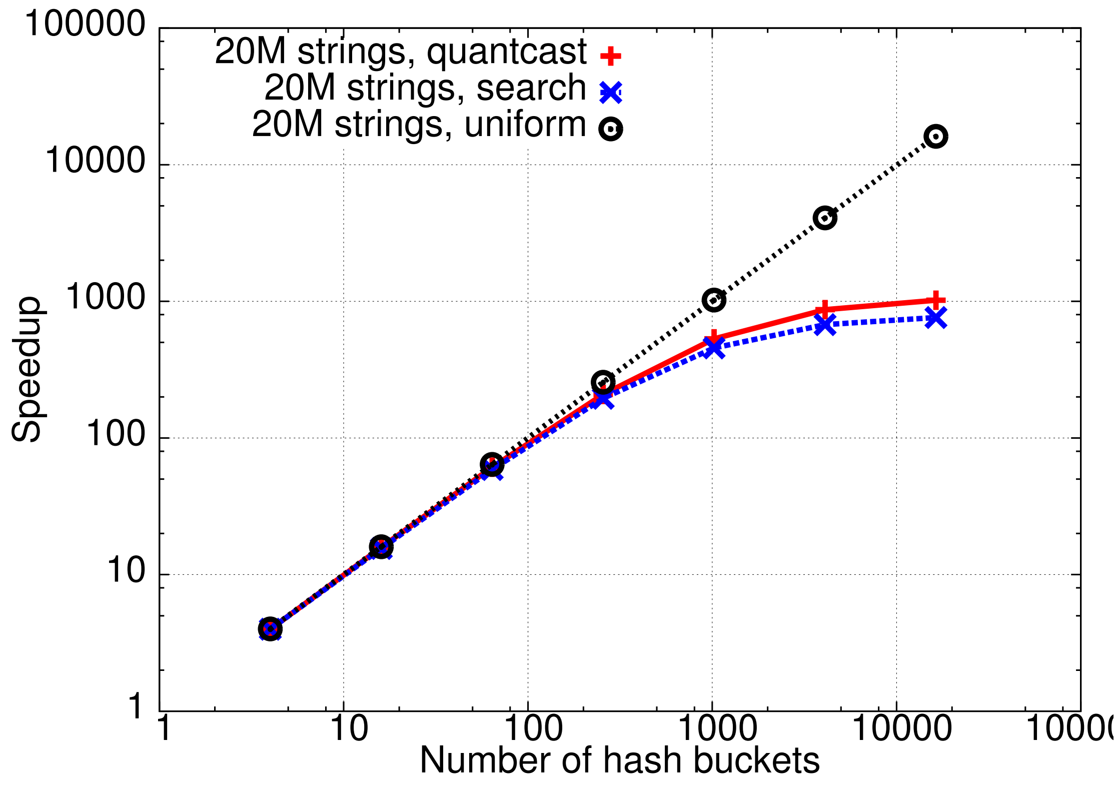


Figure 4.19: Speedup vs. number of hash buckets with 20M strings. Speedup values were similar with more strings.

speedup is not affected.

4.12.3 Microbenchmarks

We tested all operations on a Linux PC (3.1GHz CPU, 8GB RAM) and an Android smartphone (1GHz CPU, 768MB RAM). We used a string length of 100 bytes, including the padding. This length can be adjusted based on the string type and is not a fundamental limit, unlike previous work assuming 32-bit strings [83, 90]. Applebaum et al. use 1024-bit keys for encryption and ZKPs.

Table 4.3: Client microbenchmarks. OT assumes 32-bit strings. Other operations assume 100-byte strings.

	Device (language)	Operation	Ops/sec
Applebaum et al. [83]	PC (JavaScript)	Encryption (El Gamal)	21.54
		ZKP generation (GM [114])	3.22
	Smartphone (JavaScript)	Encryption (El Gamal)	0.52
		ZKP gen. (GM [114])	0.08
	PC (C)	OT (primitive, single)	0.34
		OT (batch size=25)	0.14
		OT (batch size=50)	0.14
		OT (batch size=100)	0.13
Our system	PC (JavaScript)	Split	361,627
		Join	1,181,512
		SHA-1	118,959
	Smartphone (JavaScript)	Split	2,922
		Join	22,695
		SHA-1	1,761

4.12.3.1 Computation Overhead

Our client overhead is several orders of magnitude less than Applebaum et al.'s (Table 4.3). Multiple strings can be sent in one batched OT, but heavy crypto operations hinder better performance. Our server operations are also much faster: our *PXH* operation with SHA-2 can be executed about 0.8M times/second (Table 4.4).

4.12.3.2 Memory Overhead

Assuming that the batched OT in Applebaum et al.'s system can handle 100-byte strings, the overhead depends on the batch size (Table 4.5). Even if the clients can handle the load, the proxy becomes the bottleneck: a batch size of 50 requires about 9.5GB of RAM for 1K concurrent clients. Note that a smaller batch greatly reduces the batched OT's efficiency. Our overhead is significantly lower for both the client and proxy.

Table 4.4: Server microbenchmarks. OT assumes 32-bit strings. Other operations assume 100-byte strings.

	Component (language)	Operation	Ops/sec
Applebaum et al. [83]	Database (Java)	Decryption (El Gamal)	270.20
		ZKP verification (GM [114])	19.23
	Proxy (C)	OT (batch size=25)	1.27
		OT (batch size=50)	1.21
		OT (batch size=100)	1.00
Our system	Proxy (Java)	XOR	8,300,335
		SHA-2	817,003
	Aggregator (Java)	Split	1,819,459
		Join/XOR	8,300,335
		SHA-2	817,003

	Operation	**Client**	**Proxy (1K clients)**
Applebaum et al. [83]	OT (batch size=25)	4.88 MB	4.77 GB
	OT (batch size=50)	9.77 MB	9.54 GB
	OT (batch size=100)	19.53 MB	19.07 GB
Our system	Split (25 strings)	0.005 MB	4.86 MB
	Split (50 strings)	0.01 MB	9.73 MB
	Split (100 strings)	0.02 MB	19.45 MB

Table 4.5: Memory overheads for privacy-preserving string discovery. We *optimistically* assume OT can support 100-byte strings.

4.12.4 Experiments with Real-world Data

This section shows the computational and bandwidth overheads with real-world data. generated from the distributions of two real-world datasets.

4.12.4.1 Setup

For our experiments, we vary the number of strings from 10M to 100M. The discovery threshold is set to 100. We assume each client sends 50 100-byte long strings, except for Applebaum et al.'s batched OT operation at the proxy, which uses 32-bit strings.

Applebaum et al. use a ZKP *per string* for count accuracy. It is unclear how such ZKPs would detect malicious clients *sending the same string multiple times*, but we optimistically assume each client would send *only one* ZKP for *all* its strings. In our system, the split id and seed are 16 bytes each. The string type, epoch end time and ϵ are 8 bytes each. We use both our heuristics: For the SICF optimization, we use sample sizes determined with 99% confidence level and 3% margin of error as well as a p value of 20. For the short hashes optimization, we evaluate two values for the number of buckets used by the clients (64 and 128). We set the discovery threshold to be 250.

4.12.4.2 Computation Overhead

Figure 4.20 shows the results. For Applebaum et al., we plot the CPU times for the proxy to handle the batched OT of client strings, and for the database to decrypt the OTed strings and verify the ZKPs. For our system, we plot the total CPU time for all PXH operations (counting and duplicate detection) for the aggregator, because it is used in both mirror operations.

In Applebaum et al.'s system, the proxy is clearly the bottleneck compared to the database. Our system requires at most half the CPU time of the database, and even one order of magnitude less than the proxy, while discovering almost all (99.995%) discoverable strings.

4.12.4.3 Bandwidth Overhead

In Applebaum et al.'s system, a client runs batched OT with the proxy, and sends blinded (encrypted) strings, their values, double-encrypted strings, and one ZKP. *Excluding OT*, the total cost is about 19KB for 50 strings. A client in our system uses about 10KB, and 0.12KB per poll per string type for initialization.

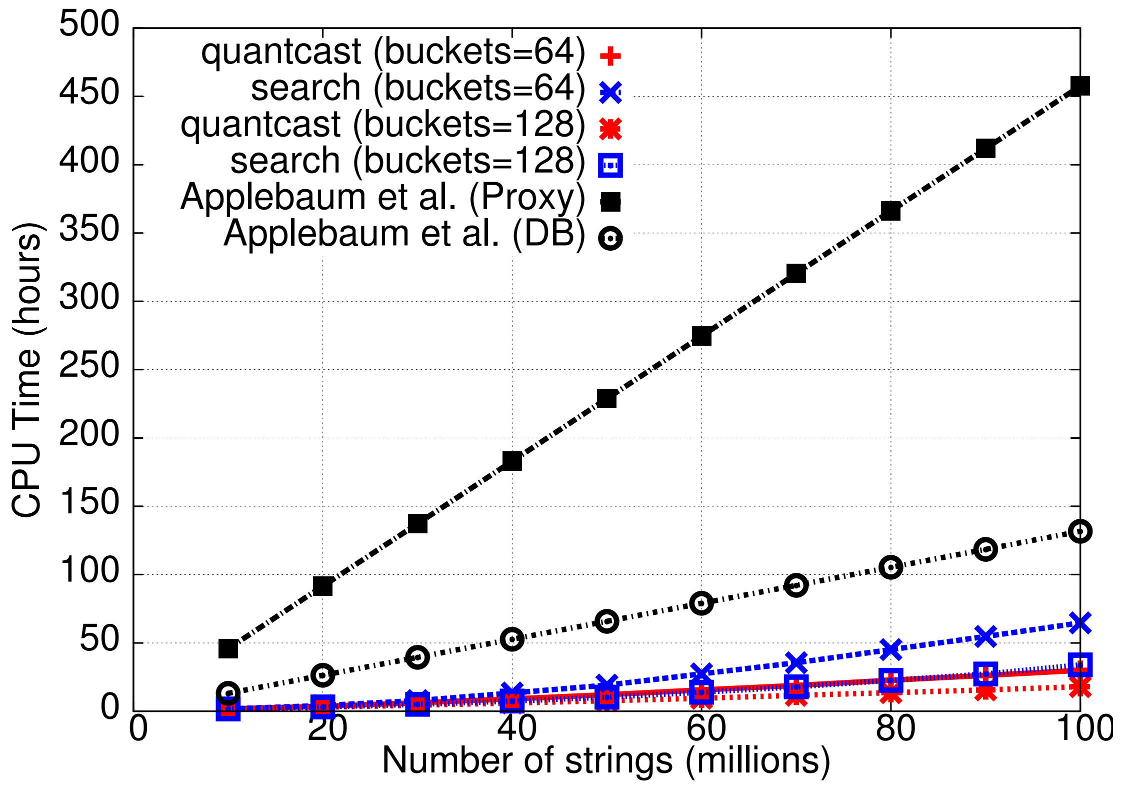


Figure 4.20: CPU time vs. number of strings.

In Applebaum et al.'s system, the costs for the proxy and database to handle 50M strings are about 36GB and 18GB, respectively, again excluding the batched OT operations for the proxy. Our system's biggest bandwidth cost is due to the *PXH* operations for counting: For 50M strings and 128 hash buckets, the aggregator's cost is about 625GB and 850GB for Quantcast and search datasets, respectively, with an average bandwidth of about 196Mbps (each proxy's cost is half). The cost of all roles in all other phases for each dataset is 43.7GB for the aggregator and 71.7GB for each proxy. The aggregator's monetary cost for bandwidth would be less than $110 in EC2 [5], and even less if the servers were in the same datacenter (run by different entities).

4.13 Conclusion & Future Work

We presented a privacy-preserving string discovery system that provides analysts with previously unknown strings of many types and their noisy counts. These strings can then be used as potential answer values for queries used in our analytics system as well as other analytics systems. Our system also detects and limits malicious clients that send duplicate strings and try to manipulate string counts.

We analyzed our system's privacy properties and evaluated its feasibility with real world data. Compared to the closest previous private aggregation system, our system reduces computation overhead several orders of magnitude for the clients, and by at least two times for the servers. Using XOR as its crypto primitive along with a low-cost blind comparison method, our system trades off cheap server bandwidth to support resource-constrained client devices and to count client strings without revealing them. We think this trade-off will be more important as the prevalence of mobile devices keeps increasing on the web.

In the future, we plan to better understand the trade-off between privacy and computation overhead in our optimizations. In particular, we will investigate the possibility of using differential privacy mechanisms within them.

Chapter 5

Literature Survey

This chapter presents the related work most relevant to the approaches presented in this thesis.

5.1 Third-party Tracking

The existing cookie policies in major browsers have been discussed in Section 2.1. None of these policies incorporate user interaction. The most similar policy to our policy is to only accept cookies from visited (i.e., as first-party) sites. However, this policy does not make a distinction at an element level. As a result, a visited site (e.g., OSN provider) can track the user activity via social widgets. On the other hand, in our policy, the third party is not allowed to learn about the user's activity unless the user chooses so and only on the pages she activates such widgets.

Of course, these third parties could try to bypass the cookie policy by not utilizing cookies at all. Browser fingerprinting is such a technique in which a signature of the browser environment is created. The browser environment includes, but is not limited to the operating system, plugins, fonts and screen resolution. The signature can then be used to identify browsers uniquely without ever storing cookies on the user's browser [106, 131]. Recent studies show that this practice is gaining popularity for third party tracking [76, 77, 137]; however, they are still not as prevalent as third party cookies are [107]. Other researchers have been working on potential defense mechanisms [111, 136]. These mechanisms are orthogonal to our cookie policy.

Ghostery [23] and Disconnect [14] are two popular browser extensions. These extensions use a blacklist of known trackers. Whenever a known tracker is found in a page, that element is not loaded. These blacklists need to be maintained periodically. Ghostery updates its blacklist every few weeks with regular extension updates whereas Disconnect checks once a day for new trackers. The disadvantages of these blacklists have been discussed in Section 2.1.

Other browser extensions such as adblockers [2–4] also utilize a blacklist of known advertisements. The default filters usually hide advertisements only after they are loaded and the user's cookie values have been sent to third party trackers. Some custom filters may specifically prevent known trackers; however, these tools still suffer from the disadvantages of a blacklist.

The most relevant browser extensions to our policy are ShareMeNot [53, 144] and Priv3 [43, 100]. These extensions have the single, specific goal of preventing online social network (OSN) providers from tracking users via social widgets. ShareMeNot reloads the entire page when the user decides to interact with a social widget. Priv3, on the other hand, utilizes a selective reload mechanism for interacted social widgets, which we also use in our implementation. In addition, we inherit Priv3's functionality of preventing third-party scripts from accessing cookie values until the user interacts with the third-party content (Section 2.5.6). Both of these extensions use a blacklist that consists of a handful of OSN widgets, whereas our policy is general and prevents tracking not only by OSN providers, but also data aggregators and advertisement networks.

Firefox's tracking protection [61, 123] is a fairly recent mechanism that has been implemented in Mozilla Firefox 35 and newer versions. Rather than relying on browser add-ons like Ghostery and Disconnect, Firefox has a built-in system for blocking known trackers. However, this mechanism still uses a subset (about 1500) of blacklisted domains obtained from Disconnect [123]. To minimize the effects of an incomplete and incorrect blacklist, Firefox updates this list every 45 minutes. Unfortunately, this blocking of blacklisted third parties also breaks the functionality of social widgets, if OSN providers are in the list.

Lightbeam from Firefox [34] dynamically tracks the third parties the browser is contacting and visualizes them for the user. These third parties then can be blocked by the user manually. Similar to the tracking protection by Firefox discussed above, such blocking breaks the functionality of social widgets.

Privacy Badger from the Electronic Frontier Foundation (EFF) [46] also dynamically tracks the third parties the browser is contacting. Privacy Badger aims to enforce that the aggregators honor the Do-Not-Track header set by browsers. To achieve this goal, Privacy Badger uses heuristics to detect and then block third parties that appear to be tracking the user without her consent. If the content is deemed necessary for the page, cookies are removed from these requests. More recently, Privacy Badger started also preventing tracking by social widgets. However, it uses the same blacklist as ShareMeNot [53] for a few OSN sites (i.e., AddThis, Facebook, Google, LinkedIn, Pinterest, Stumbleupon and Twitter), and the interaction with the widget is blocked if not manually overridden.

Pan et al. [138] propose a browser, TrackingFree, in which the client data is partitioned into isolated profiles that are generated on the fly and assigned a new principal for each site the user visits. The third party content in each site, including set cookies, would be present in different profiles. As a result, the third parties will not be able to receive the same identifiers (i.e., cookies) to track the user across different sites. To achieve its goal, TrackingFree has to maintain a complex state of principals governed by a sophisticated algorithm. Furthermore, it creates user experience problems. For example, if the user wants to share three different links from three different sites on her Facebook account, she would need to login to Facebook three times [138]. Although synchronization among principals can be enabled by the user to avoid this 'multiple login' situation, it also opens the possibility for the third party (in this case, Facebook) to track the user after the synchronization.

Kontaxis et al. [124] propose a technique for preserving personalization functionality of social widgets while preserving user privacy. In this technique, the browser keeps a local copy of the personalized content, such that a social widget can be rendered without ever contacting the OSN provider. This private information is gathered when the user naturally visits the OSN. For example, to enable Facebook Like buttons with personalized content, the client gathers information about a user's friends and their "likes" and stores them locally. When a social widget is encountered, this information is used to render the widget with the personalized content that would have been available if the client were to contact the OSN. Any other information is gathered from publicly available information from the OSN. The authors also present a functional prototype specialized to Facebook.

This approach can preserve more personalized content compared with our cookie policy: our two-click control renders the personalized content only after the user decides to interact with the widget. Otherwise, it shows publicly available information (e.g., the total number of "likes"). This seamless personalization by the above technique, however, has the following shortcomings. First, the client software needs to recognize all OSN providers, such that it can gather the required information for personalization. This need essentially means that the developers have to create a custom scraper for each OSN and maintain them to adapt any changes the OSN provider makes to its pages. Second, to efficiently gather the personalized information, this scraping process requires some API support from the OSN provider, which may not provide full functionality or not exist at all.

5.2 Web Analytics

Although web analytics, as far as we know, has never been considered in a privacy context, there have been a number of approaches for providing users with privacy guarantees in distributed settings. These approaches differ from our system in terms of assumptions, requirements and goals. Here, we review past work most related to our system, in the areas of anonymous communication, privacy-preserving aggregation and differential privacy.

Users can use a VPN proxy or an anonymizing network like TOR [59] for anonymous communication. While providing privacy benefits for anonymous browsing, these systems are not suitable for non-tracking web analytics: they may violate our non-tracking goal (e.g., a VPN proxy observing the source and the destination of a session), mislead the publisher to collect incorrect information (e.g., the proxy's or TOR exit node's address misleading a publisher using IP geolocation), or most importantly, do not add any noise to the results (e.g., if sensitive data is collected).

For these reasons, researchers have proposed systems that both preserve users' privacy and enable accurate collection and aggregation of private information [135, 140]. Anonygator [140] privately aggregates pre-defined histograms (i.e., ranges) from distributed user devices, but assumes the shared data will not leak privacy. P3 [135] is a privacy-preserving, distributed personalization system, but requires a method to determine which data is safe to supply for personalization. These systems, however, either make assumptions about the collected information (i.e., that it will not to leak the source identity) [140], or require an algorithm to decide which data are safe to contribute, which may not be easy to devise [135]. In contrast, our system combines differentially-private noise and separate encryption of answer messages to protect against identity leakage through the aggregated data, without any assumptions or prerequisites. Furthermore, these systems rely on an anonymity network, such as TOR, to hide the source identity (i.e., IP address), whereas our system utilizes an already existing entity (i.e., the publisher) as an anonymizing proxy.

Applebaum et al.'s system [83] described in Section 4.12 utilizes a proxy and a database for privacy-preserving aggregation of participants' private data in the form of {key, value} pairs. The system's main goal is to achieve this aggregation without exposing participants' keys to each other, which makes it more relevant to our string discovery system. Nevertheless, the strong cryptographic model the system is using causes each participant higher overhead than in our system, such that applying it in our web analytics setting would not be very efficient. Therefore, this system is perhaps more suitable for publishers wanting to share their own, already aggregated analytics data rather than for users.

While these systems provide users with *some* privacy guarantees, they do not utilize any differential privacy mechanisms, which are considered to give stronger and more formal guarantees than existing techniques [102,103,105]. Many original uses of differential privacy, however, assume the existence of a central database controlling the disclosure of results [85,118]. Although attempts have been made to provide differential privacy in a distributed environment, these attempts either incur high overhead [104] or suffer from client churn [142,147], making them impractical in a large-scale environment.

To tackle this practicality problem, recent proposals employ different approaches for generating differentially-private noise. Duan et al. propose a system called P4P that utilizes two honest-but-curious (HbC) *servers to add noise* [101]. They employ relatively efficient, but still costly zero-knowledge proofs to ensure accuracy of the aggregated result. Hardt and Nath also use two HbC servers, but propose that *users add noise*, such that honest users compensate for the noise that unavailable or malicious users did not generate [117]. While preserving honest users' privacy, this system allows a malicious user to distort the result arbitrarily. Neither of these systems is suitable for web analytics, assuming that these two servers would correspond to the publisher and the data aggregator: letting a server know about the other server for a given client violates our non-tracking goal.

More recently, Chen et al. proposed a proxy-based system (PDDP) for achieving differential privacy in a distributed environment [98]. PDDP utilizes *only one* HbC proxy that distributes an analyst's queries to clients, collects responses, and adds noise in a *blind* fashion, such that it does not know how much noise it added. PDDP does not scale for web analytics purposes for two reasons. First, PDDP has no way of selecting different groups of users to receive a given query. In our setting, these groups correspond to the users that visit a given website. If the proxy knew these separate groups (i.e., knowing a user is visiting a particular website), it would violate our non-tracking goal. Our system exploits publishers for this purpose because publishers inherently know which users visit their websites.

Second, PDDP encrypts every bucket answer, 'yes' and 'no' alike, making it very costly for the large-bucket queries that are needed in web analytics. This encryption approach also prevents PDDP to overcome its first shortcoming of distinguishing different user groups: to achieve the same non-tracking property as our system, PDDP would need to distribute *all* queries to *all* users and collect *all* of their answers.

5.3 Privacy-preserving String Discovery

XOR [94,148,152] and matrix multiplication [122] are used as lightweight primitives for anonymous communication instead of relatively expensive public key cryptography operations. These systems do not aggregate user data privately.

Most database privacy research assumes a trusted database [102,129,150]. We refer the readers to a survey [113]. Perhaps, the most relevant systems are the following two. Chen et al. [96] publish sequential data via variable-length n-grams with differential privacy. McSherry et al. [132] discover common payloads in network traces by choosing strings via their noisy counts and iteratively increasing their lengths. Like the previous systems, these systems also assume a centralized database. Unlike these systems, we assume a distributed setting, in which user data resides on user devices.

To reduce the trust in the database, some systems encrypt user data before storing it [139,146,149]. Afterwards, such data can be searched and queried. In these systems, however, the results (i.e., counts) are not noisy. As a result, they may allow a malicious analyst to learn sensitive information.

Trust in the storage entity can also be decreased using multiple databases. For privacy-preserving queries over multiple databases, Chow et al. [99] propose a two-entity model. In this model, one entity shares a secret with the databases to obfuscate results, while the other aggregates obfuscated data. This scheme is similar to the aggregator sharing a secret with the clients, who then use the secret as a salt to hash their strings. The hashes are then used to count client strings. However, if the secret is shared with the aggregating entity, for example, if the aggregator in our setting runs fake clients, the privacy properties are lost. In contrast, our threat model allows components to run fake clients. Furthermore, Chow et al. assume that a database (i.e., a client in our setting) supplies correct data, which may not be true in analytics scenarios. Our system utilizes a duplicate detection mechanism to limit the effect of these clients.

Sepia [90] is a secure multiparty computation (SMC) framework that specializes in aggregation of network events without a centralized entity, and can be used for top-k queries [89]. Via optimized comparison operations, it scales better than other SMC frameworks, but is limited to short strings (i.e., length of an IP4 address) and fewer participants (i.e., <100), and thus, cannot be directly applied to our scenarios. Proposals to improve SMC performance for mobile devices [92,120,121] assume a two-party model. It is unclear how these proposals can be extended to support millions of clients for our purposes.

As described in Section 4.12 and mentioned in the previous section, Applebaum et al. propose a system with a proxy and aggregator to privately aggregate participants' private data. The main difference between our privacy-preserving string discovery system and their system is that Applebaum et al. do not add noise to the aggregated result. Lack of noise opens the possibility for an attacker to create Sybil nodes to pollute the results and gain knowledge about the existence of a particular string value. Additionally, Applebaum et al. utilize sophisticated cryptographic operations, which, as shown in Section 4.12, add significant overheads to the clients. Finally, although they utilize (expensive) zero-knowledge proofs to ensure that the participants only submit a value of '0' or '1' for a particular key, there is no mention of a malicious participant submitting the *same key* multiple times. Our system prevents this issue with our duplicate detection mechanism.

Approaches to find frequent items over distributed streams [93,126,130,153,154] have similar goals; however, we are not aware of any system that achieves all our goals at the same time in a web-scale environment (i.e., discovering unknown strings, lightweight client operations, privacy and anonymity for clients, handling malicious clients). Hsu et al. [119] describe an algorithm to privately find heavy hitters in a distributed setting, but one item at a time. By contrast, our system discovers multiple strings of one type in one run.

To address both issues about scalability and malicious clients, recent distributed DP systems employ pre-defined string values and centralized entities. πBox [128] uses pre-defined counter names. It uses a trusted platform to restrict the interface how much and how often an mobile application instance (i.e., client) can update a counter. The trusted platform also adds noise to counter values before reporting them to the developers. Hardt et al. [117] also use a counter for advertisement impressions (or clicks), but utilize the clients to add noise. Two honest-but-curious servers then aggregate the counter values. These two systems assume that the counter names are well-known. Our privacy-preserving string discovery system complements these systems if they were to require additional counter names, such as user-defined counters for different applications.

Recently, Friedman et al. [112] proposed a system to monitor distributed stream sources for frequent items with differential privacy. However, the observed items come from a fixed list enumerated by the analysts beforehand.

RAPPOR [109] uses Bloom Filters [88] and a differentially-private, randomized response scheme to obtain frequencies of client strings. However, it requires a list of candidate strings to decode the filters. RAPPOR does not handle client strings changing over time (e.g., 'recently visited sites') whereas our discovery procedures can be run periodically to tolerate such changing user data. Furthermore, RAPPOR assumes that the analyst and the aggregator are the same entities, and thus, the analyst knows all the clients. This situation is less general than our system, in which the aggregator can provide string discovery service to multiple analysts in a privacy-preserving manner.

As described previously, PDDP [98] enables aggregation of private user data by distributing queries to clients and adding noise via a centralized proxy. However, queries in PDDP also require a list of pre-defined string values as potential answers, similar to our non-tracking web analytics system described in Chapter 3 and SplitX [97]. Our string discovery system is complementary for all these systems, including PDDP in this regard.

Chapter 6

Summary & Final Remarks

Current architectures to provide essential web services for social widgets and analytics utilize methods that enable third-party service providers to track users across the web. This tracking leads to privacy concerns among users, who often block and prevent tracking related elements with client-side tools, reducing the benefits and utility of these services for the publishers. We think that solutions that favor one side in this struggle are not viable, and user privacy and functionality should be considered together. In this thesis, we presented approaches to relieve this tension between privacy for users and functionality for publishers and other service providers.

We first proposed and explored a new third-party cookie policy that prevents tracking, but also to enables social widgets on-demand. The power of this policy comes from its generalization of the simple idea of withholding third-party cookies while loading third-party resources on web pages until the user interaction. This generalization removes the need for a blacklist of tracking elements as well as alleviates associated problems of curation, maintenance and that the blacklist can be bypassed. We showed that our policy is effective in giving the users more control for social widgets by distinguishing them from advertisements with high accuracy, eliminates third-party tracking via cookies by online social network providers as well as data aggregators, and imposes low overhead.

We then presented a web analytics system that eliminated the need for third-party tracking for the purposes of extended web analytics. The system can provide publishers and aggregators with more accurate and more types of extended web analytics information by keeping user data on the user device and directly querying it via the publishers. At the same time, the system prevents the aggregator from tracking the users by utilizing the publisher as an anonymizing proxy and adding differentially-private noise to the results. We showed that this system is feasible, easy to deploy and imposes low overhead on the clients as well as publishers and acceptable overhead for the aggregators.

The above system and other similar systems require a list of string values that will be used as potential answer values for the queries they distribute. Our final contribution is a system that helps the analysts (e.g., publishers) to discover previously unknown string values for this purpose. Although the discovery system does not provide strict guarantees of differential privacy, it still enables aggregation of user data under realistic assumptions and in a privacy-preserving way , in which clients are given anonymity and unlinkability properties, and attacks to deanonymize a client's strings are made difficult. Our system trades off cheap server bandwidth for low client overhead, such that even the low-resource user devices such as mobile phones/tablets can participate. As a result, analysts can take advantage of the rich user data accumulating on these devices that are becoming more and more prevalent.

Although social widgets and web analytics play an integral role in today's web, they are certainly not the only essential services that make the web function. Advertisements have always been a part of the web ecosystem allowing website publishers to monetize their content and sustain their operations. Meanwhile, reaching the right audience and increasing the effect of the advertisements have been the ultimate goal of advertisers. Unfortunately, this goal has been in direct conflict with user privacy most of the time due to targeted advertisements that are determined via third-party tracking.

Recommendation systems are also an another important part of today's web. These systems utilize user preferences and previous consumption to provide users with suggestions for new content. The sensitivity of the information that is used for these recommendations naturally causes privacy concerns.

There have been alternative approaches considering user privacy while providing targeted advertisements [84, 115, 117, 143, 151]. Similarly, there have been attempts to provide privacy-preserving recommendations by utilizing cryptography alongside a server that operates on encrypted data [78, 86, 108]. These systems showcase again that it is possible to provide privacy and functionality at the same time for web services, and are complementary to our approaches.

The adoption of these systems, however, does not depend solely on their technical feasibility. A combination of other factors, such as the economic incentives for the players, privacy awareness of the users as well as legal frameworks supporting privacy rights, certainly play a significant role. The challenge in designing such systems then becomes not only finding the technical solutions to achieve functionality and privacy goals at the same time, but also finding the right combination of these factors to make these systems adoptable. This task is certainly not easy; however, we think that the approaches and systems presented in this thesis as well as others mentioned above are steps in the right direction.

Going forward, future applications or systems that utilize the web or a web-like medium should consider user privacy as one of their main goals; user privacy should not be an afterthought. For example, the prevalence of devices in the form of cheap sensors is increasing dramatically [1]. These devices are capable of connecting to each other as well as to the Internet and the web. Found in various areas of our lives from home automation [54, 58] to fitness applications [32] and toys [60], the Internet of Things [30] is certainly going to enable various new and exciting functionalities. However, one should not forget that the accumulation of rich amounts of user data will have privacy implications. Ideas and approaches utilized in this thesis, such as keeping user data on user devices and under users' own control as well as distributing functionality among collaborating components, can be adapted for these new systems.

Bibliography

[1] 15 Mind-blowing stats about the Internet of Things. http://www.cmo.com/articles/2015/4/13/mind-blowing-stats-internet-of-things-iot.html.

[2] Adblock - Browser faster. Ad-free. https://getadblock.com.

[3] Adblock Edge :: Add-ons for Firefox. https://addons.mozilla.org/en-US/firefox/addon/adblock-edge/.

[4] Adblock Plus - Surf the web without annoying ads! https://adblockplus.org/.

[5] Amazon EC2 Pricing. http://aws.amazon.com/ec2/pricing/.

[6] Analytics Technology Web Usage Statistics. http://trends.builtwith.com/analytics. Aug 2, 2012.

[7] AW Stats - Free log file analyzer for advanced statistics (GNU GPL). http://awstats.sourceforge.net.

[8] Bango Dashboard Analytics. http://bango.com/dashboard-analytics/.

[9] Belgium takes Facebook to court over privacy breaches and user tracking. http://www.theguardian.com/technology/2015/jun/15/

belgium-facebook-court-privacy-breaches-ads.

[10] BlueKai Consumers. http://bluekai.com/consumers_optout.php.

[11] BrightTag ONE-Click Privacy. http://www.brighttag.com/privacy/.

[12] Cookie Synching. http://www.krux.com/company_blog/cookie_synching/.

[13] Countly Mobile Analytics. http://count.ly.

[14] Disconnect. https://disconnect.me/.

[15] EasyPrivacy. https://easylist.adblockplus.org/.

[16] Facebook Ads Will Now Follow You No Matter What Device You're Using. http://www.wired.com/2014/09/facebook-launches-atlas/.

[17] Facebook Faces Privacy Lawsuit From Belgian Watchdog. http://techcrunch.com/2015/06/15/facebook-faces-privacy-lawsuit-from-belgian-watchdog/.

[18] FatCow Web Hosting. http://www.fatcow.com.

[19] Firebug. https://getfirebug.com/.

[20] Firefox Personalization Study. https://addons.mozilla.org/en-US/firefox/addon/firefox-personalization-study/.

[21] Flurry. http://www.flurry.com/.

[22] FTC Issues Final Commission Report on Protecting Consumer Privacy. http://ftc.gov/opa/2012/03/privacyframework.shtm.

[23] Ghostery. https://www.ghostery.com/.

[24] Google Public Policy Blog — Keep your opt-outs. http://googlepublicpolicy.blogspot.com/2011/01/keep-your-opt-outs.html.

[25] Google Will Pay $22.5 Million to Settle FTC Charges it Misrepresented Privacy Assurances to Users of Apple's Safari Internet Browser. http://www.ftc.gov/news-events/press-releases/2012/08/google-will-pay-225-million-settle-ftc-charges-it-misrepresented.

[26] grindtv.com Site Overview. http://www.alexa.com/siteinfo/grindtv.com. July 10, 2015.

[27] Grindtv.com Traffic and Demographic Statistics by Quantast. https://www.quantcast.com/grindtv.com/demographics/web. July 10, 2015.

[28] Home of the Webalizer. http://www.webalizer.org/.

[29] Internet Explorer 9 Tracking Protection Lists. http://ie.microsoft.com/testdrive/Browser/TrackingProtectionLists/faq.html.

[30] Internet of Things - Cisco Systems. http://www.cisco.com/web/solutions/trends/iot/overview.html.

[31] iPage Web Hosting. http://www.ipage.com.

[32] Jawbones New Wristband Adds You to the Internet of Things. http://www.technologyreview.com/news/521606/jawbones-new-wristband-adds-you-to-the-internet-of-things/.

[33] Lawsuit accuses comScore of extensive privacy violations. http://www.computerworld.com/s/article/9219444/Lawsuit_accuses_comScore_of_extensive_privacy_violations.

[34] Lightbeam Firefox Add-on. https://addons.mozilla.org/en-US/firefox/addon/lightbeam/.

[35] Localytics Mobile App Marketing and App Analytics Blog. http://www.localytics.com/.

[36] Mint Analytics. http://haveamint.com/.

[37] Mobclix: Mobile Application Store offers the Best Iphone Apps & Solutions. http://www.mobclix.com/.

[38] Mobile App Analytics - Google Analytics. https://www.google.com/analytics/mobile/.

[39] mod_alias - Apache HTTP Server Version 2.2. https://httpd.apache.org/docs/2.2/mod/mod_alias.html.

[40] Open Web Analytics. http://openwebanalytics.com.

[41] Piwik Web Analytics. http://piwik.org.

[42] Piwik Web Hosting. http://www.arvixe.com/piwik_hosting.

[43] Priv3 Firefox Add-on. https://addons.mozilla.org/en-US/firefox/addon/priv3/.

[44] Priv3+ Firefox Add-on. https://addons.mozilla.org/en-US/firefox/addon/priv3plus/.

[45] Priv3+ Google Chrome Extension. https://chrome.google.com/webstore/detail/priv3%20/oigbhpafgooddcnlapndedpakbgpopoc.

[46] Privacy Badger — Electronic Frontier Foundation. https://www.eff.org/privacybadger.

[47] Privacy Lawsuit Targets Net Giants Over 'Zombie' Cookies. `http://www.wired.com/threatlevel/2010/07/zombie-cookies-lawsuit`.

[48] ProVerif. `http://proverif.inria.fr/`.

[49] Quantcast Clearspring Flash Cookie Class Action Settlement. `http://www.topclassactions.com/lawsuit-settlements/lawsuit-news/920`.

[50] Quantcast Opt-Out. `http://www.quantcast.com/how-we-do-it/consumer-choice/opt-out/`.

[51] Quantcast Top Ranking International Websites. `https://www.quantcast.com/top-sites`.

[52] Safari Adds Do Not Track Features. `http://mashable.com/2011/04/14/safari-do-not-track`.

[53] ShareMeNot Firefox Add-on. `https://addons.mozilla.org/en-US/firefox/addon/sharemenot/`.

[54] SmartThings — Smart Home. Intelligent Living. `http://www.smartthings.com/`.

[55] The Do Not Track Option: Giving Consumers a Choice. `http://www.ftc.gov/opa/reporter/privacy/donottrack.shtml`.

[56] The hidden perils of cookie syncing. `https://freedom-to-tinker.com/blog/englehardt/the-hidden-perils-of-cookie-syncing/`.

[57] The Mozilla Blog — Mozilla Firefox 4 Beta, now including "Do Not Track" capabilities. `http://blog.mozilla.com/blog/2011/02/08`.

[58] The scattered, futuristic world of home automation. `http://edition.cnn.com/2013/01/12/tech/innovation/future-home-automation/`.

[59] Tor Project. https://www.torproject.org/.

[60] Toymail - Keeping families connected without more screen time. http://www.toymail.co/.

[61] Tracking Protection on Firefox. https://support.mozilla.org/en-US/kb/tracking-protection-firefox.

[62] Usage Statistics and Market Share of Traffic Analysis Tools. http://w3techs.com/technologies/overview/traffic_analysis/all. Aug 2, 2012.

[63] User Personalization Update. https://blog.mozilla.org/labs/2013/12/user-personalization-update/.

[64] usnews.com Site Overview. http://www.alexa.com/siteinfo/usnews.com. July 10, 2015.

[65] Usnews.com Traffic and Demographic Statistics by Quantast. https://www.quantcast.com/usnews.com/demographics/web. July 10, 2015.

[66] W3 - BlueKai Proposal for Browser Based Do-Not-Track Functionality. http://www.w3.org/2011/track-privacy/papers/BlueKai.pdf.

[67] W3Perl Free Log File Analyzer. http://www.w3perl.com.

[68] Web Tracking Protection. http://www.w3.org/Submission/web-tracking-protection/.

[69] Widget Technologies Web Usage Statistics. http://trends.builtwith.com/widgets.

[70] Abine. http://www.abine.com/, last access on October 8, 2014.

[71] ComScore: Mobile Will Force Desktop Into Its Twilight In 2014. http://www.businessinsider.com/

`mobile-will-eclipse-desktop-by-2014-2012-6`, last access on October 8, 2014.

[72] Internet Access Statistics. `http://www.ons.gov.uk/ons/dcp171778_301822.pdf`, last access on October 8, 2014.

[73] Mobile Internet traffic gaining fast on desktop Internet traffic. `http://news.cnet.com/8301-1023_3-57556943-93/`, last access on October 8, 2014.

[74] Martín Abadi and Cédric Fournet. Mobile Values, New Names, and Secure Communication. In *POPL*, 2001.

[75] Masayuki Abe and Eiichiro Fujisaki. How to date blind signatures. In *Advances in Cryptology - ASIACRYPT '96, International Conference on the Theory and Applications of Cryptology and Information Security, Kyongju, Korea, November 3-7, 1996, Proceedings*, 1996.

[76] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The Web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 674–689. ACM, 2014.

[77] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. FPDetective: Dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1129–1140. ACM, 2013.

[78] Esma Aïmeur, Gilles Brassard, José M Fernandez, and Flavien Serge Mani Onana. Alambic: a privacy-preserving recommender system for electronic commerce. *International Journal of Information Security*, 7(5):307–334, 2008.

[79] Istemi Ekin Akkus. Formal Model and Verification of Privacy-preserving String Discovery for Mobile and Web Analytics. In *Max Planck Institute for Software Systems Technical Report MPI-SWS-2014-006*, 2014.

[80] Istemi Ekin Akkus, Ruichuan Chen, and Paul Francis. String Discovery for Private Analytics. In *Max Planck Institute for Software Systems Technical Report MPI-SWS-2013-006*, 2013.

[81] Istemi Ekin Akkus, Ruichuan Chen, Michaela Hardt, Paul Francis, and Johannes Gehrke. Non-tracking Web Analytics. In *CCS*, 2012.

[82] Istemi Ekin Akkus and Nicholas Weaver. The Case for a General and Interaction-based Third-party Cookie Policy. In *Proceedings of the IEEE Workshop on Web 2.0 Security and Privacy (W2SP)*, 2015.

[83] Benny Applebaum, Haakon Ringberg, Michael J. Freedman, Matthew Caesar, and Jennifer Rexford. Collaborative, Privacy-Preserving Data Aggregation at Scale. In *PETS*, 2010.

[84] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. ObliviAd: Provably Secure and Practical Online Behavioral Advertising. In *IEEE Symposium on Security and Privacy*, 2012.

[85] Boaz Barak, Kamalika Chaudhuri, Cynthia Dwork, Satyen Kale, Frank McSherry, and Kunal Talwar. Privacy, accuracy, and consistency too: a holistic solution to contingency table release. In *PODS*, 2007.

[86] Anirban Basu, Jaideep Vaidya, and Hiroaki Kikuchi. Efficient privacy-preserving collaborative filtering based on the weighted slope one predictor. *Journal of Internet Services and Information Security (JISIS)*, 1(4):26–46, 2011.

[87] Arnar Birgisson, Frank McSherry, and Martín Abadi. Differential privacy with information flow control. In *PLAS*, 2011.

[88] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13:422–426, 1970.

[89] Martin Burkhart and Xenofontas A. Dimitropoulos. Fast Privacy-Preserving Top-k Queries Using Secret Sharing. In *ICCCN*, 2010.

[90] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas A. Dimitropoulos. SEPIA: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics. In *USENIX Security Symposium*, 2010.

[91] Tianjie Cao, Dongdai Lin, and Rui Xue. A Randomized RSA-based Partially Blind Signature Scheme for Electronic Cash. *Computers & Security*, 24(1), 2005.

[92] Henry Carter, Benjamin Mood, Patrick Traynor, and Kevin Butler. Secure Outsourced Garbled Circuit Evaluation for Mobile Devices. In *USENIX Security Symposium*, 2013.

[93] T.-H. Hubert Chan, Mingfei Li, Elaine Shi, and Wenchang Xu. Differentially Private Continual Monitoring of Heavy Hitters from Distributed Streams. In *PETS*, 2012.

[94] David Chaum. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *J. Cryptology*, 1(1):65–75, 1988.

[95] David L. Chaum. Blind Signatures for Untraceable Payments. *Advances in Cryptology (CRYPTO)*, 1982.

[96] Rui Chen, Gergely Ács, and Claude Castelluccia. Differentially private sequential data publication via variable-length n-grams. In *CCS*, 2012.

[97] Ruichuan Chen, Istemi Ekin Akkus, and Paul Francis. SplitX: High-Performance Private Analytics. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2013.

[98] Ruichuan Chen, Alexey Reznichenko, Paul Francis, and Johannes Gehrke. Towards Statistical Queries over Distributed Private User Data. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[99] Sherman S. M. Chow, Jie-Han Lee, and Lakshminarayanan Subramanian. Two-Party Computation Model for Privacy-Preserving Queries over Distributed Databases. In *Network and Distributed System Security (NDSS) Symposium*, 2009.

[100] Mohan Dhawan, Christian Kreibich, and Nicholas Weaver. Priv3: A third party cookie policy. In *W3C Workshop: Do Not Track and Beyond*, 2012.

[101] Yitao Duan, John Canny, and Justin Z. Zhan. P4P: Practical Large-Scale Privacy-Preserving Distributed Computation Robust against Malicious Users. In *USENIX Security Symposium*, pages 207–222, 2010.

[102] Cynthia Dwork. Differential Privacy. In *ICALP*, 2006.

[103] Cynthia Dwork. Differential Privacy: A Survey of Results. In *TAMC*, pages 1–19, 2008.

[104] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our Data, Ourselves: Privacy Via Distributed Noise Generation. In *EUROCRYPT*, pages 486–503, 2006.

[105] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating Noise to Sensitivity in Private Data Analysis. In *TCC*, pages 265–284, 2006.

[106] Peter Eckersley. How unique is your web browser? In *Privacy Enhancing Technologies*, 2010.

[107] Steven Englehardt, Dillon Reisman, Christian Eubank, Peter Zimmerman, Jonathan Mayer, Arvind Narayanan, and Edward W Felten. Cookies that give you away: The surveillance implications of web tracking. In *Proceedings of the 24th International Conference on World Wide Web*, 2015.

[108] Zekeriya Erkin, Michael Beye, Thijs Veugen, and Reginald L Lagendijk. Efficiently computing private recommendations. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5864–5867. IEEE, 2011.

[109] Ùlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *CCS*, 2014.

[110] Ziba Eslami and Mehdi Talebi. A New Untraceable Off-line Electronic Cash System. *Electronic Commerce Research and Applications*, 10(1):59–66, 2011.

[111] Amin FaizKhademi, Mohammad Zulkernine, and Komminist Weldemariam. FP-Guard: Detection and Prevention of Browser Fingerprinting. In *Data and Applications Security and Privacy XXIX*, pages 293–308. Springer, 2015.

[112] Arik Friedman, Izchak Sharfman, Daniel Keren, and Assaf Schuster. Privacy-Preserving Distributed Stream Monitoring. In *Network and Distributed System Security (NDSS) Symposium*, 2014.

[113] Benjamin C. M. Fung, Ke Wang, Rui Chen, and Philip S. Yu. Privacy-preserving data publishing: A survey of recent developments. *ACM Comput. Surv.*, 42(4), 2010.

[114] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on computing*, 18(1):186–208, 1989.

[115] Saikat Guha, Bin Cheng, and Paul Francis. Privad: Practical Privacy in Online Advertising. In *USENIX Symposium on Networked Systems Design and Implementation*

(NSDI), 2011.

[116] Saikat Guha, Mudit Jain, and Venkata N Padmanabhan. Koi: A location-privacy platform for smartphone apps. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[117] Michaela Hardt and Suman Nath. Privacy-aware personalization for mobile advertising. In *CCS*, 2012.

[118] Michael Hay, Vibhor Rastogi, Gerome Miklau, and Dan Suciu. Boosting the accuracy of differentially private histograms through consistency. *Proc. VLDB Endow.*, 3(1-2), September 2010.

[119] Justin Hsu, Sanjeev Khanna, and Aaron Roth. Distributed Private Heavy Hitters. In *ICALP*, 2012.

[120] Yan Huang, Peter Chapman, and David Evans. Privacy-preserving applications on smartphones. *USENIX Workshop on Hot Topics in Security*, 2011.

[121] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster Secure Two-Party Computation Using Garbled Circuits. *USENIX Security Symposium*, 2011.

[122] Sachin Katti, Jeff Cohen, and Dina Katabi. Information Slicing: Anonymity Using Unreliable Overlays. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[123] Georgios Kontaxis and Monica Chew. Tracking protection in firefox for privacy and performance. In *Proceedings of the IEEE Workshop on Web 2.0 Security and Privacy (W2SP)*, 2015.

[124] Georgios Kontaxis, Michalis Polychronakis, Angelos D Keromytis, and Evangelos P Markatos. Privacy-preserving social plugins. In *Proceedings of the 21st USENIX Security Symposium)*, 2012.

[125] Aleksandra Korolova, Krishnaram Kenthapadi, Nina Mishra, and Alexandros Ntoulas. Releasing search queries and clicks privately. In *WWW*, pages 171–180, 2009.

[126] Abhishek Kumar and Jun Xu. Sketch Guided Sampling-Using On-Line Estimates of Flow Size for Adaptive Data Collection. In *INFOCOM*, 2006.

[127] Ralf Küsters and Tomasz Truderung. Reducing Protocol Analysis with XOR to the XOR-free Case in the Horn Theory Based Approach. In *CCS*, 2008.

[128] Sangmin Lee, Edmund L Wong, Deepak Goel, Mike Dahlin, and Vitaly Shmatikov. πBox: a platform for privacy-preserving apps. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[129] Ashwin Machanavajjhala, Johannes Gehrke, Daniel Kifer, and Muthuramakrishnan Venkitasubramaniam. l-Diversity: Privacy Beyond k-Anonymity. In *ICDE*, page 24, 2006.

[130] Gurmeet Singh Manku and Rajeev Motwani. Approximate Frequency Counts over Data Streams. In *VLDB*, 2002.

[131] Jonathan R Mayer. Any person... a pamphleteer. *Senior Thesis, Stanford University*, 2009.

[132] Frank McSherry and Ratul Mahajan. Differentially-private network trace analysis. In *SIGCOMM*, pages 123–134, 2010.

[133] Frank D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, 2009.

[134] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[135] Animesh Nandi, Armen Aghasaryan, and Makram Bouzid. P3: A Privacy Preserving Personalization Middleware for Recommendation-based Services. In *HotPETS*, 2011.

[136] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. Privaricator: Deceiving fingerprinters with little white lies. In *Proceedings of the 24th International Conference on World Wide Web (WWW)*, 2015.

[137] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013.

[138] Xiang Pan, Yinzhi Cao, and Yan Chen. I do not know what you visited last summer: Protecting users from third-party web tracking with trackingfree browser. In *Network and Distributed System Security (NDSS) Symposium*, 2015.

[139] Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[140] Krishna P. N. Puttaswamy, Ranjita Bhagwan, and Venkata N. Padmanabhan. Anonygator: Privacy and Integrity Preserving Data Aggregation. In *International Conference on Middleware*, 2010.

[141] Michael O Rabin. How to exchange secrets by oblivious transfer. Technical report, TR-81, Harvard Aiken Computation Laboratory, 1981.

[142] Vibhor Rastogi and Suman Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *SIGMOD Conference*, pages 735–746, 2010.

[143] Alexey Reznichenko and Paul Francis. Private-by-Design Advertising Meets the Real World. In *CCS*, 2014.

[144] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and Defending Against Third-Party Tracking on the Web. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[145] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1), 2003.

[146] Elaine Shi, John Bethencourt, Hubert T.-H. Chan, Dawn Xiaodong Song, and Adrian Perrig. Multi-Dimensional Range Query over Encrypted Data. In *IEEE Symposium on Security and Privacy*, 2007.

[147] Elaine Shi, T.-H. Hubert Chan, Eleanor G. Rieffel, Richard Chow, and Dawn Song. Privacy-Preserving Aggregation of Time-Series Data. In *Network and Distributed System Security (NDSS) Symposium*, 2011.

[148] Emin Gün Sirer, Sharad Goel, Mark Robson, and Dogan Engin. Eluding carnivores: file sharing with strong anonymity. In *ACM SIGOPS European Workshop*, 2004.

[149] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical Techniques for Searches on Encrypted Data. In *IEEE Symposium on Security and Privacy*, 2000.

[150] Latanya Sweeney. k-Anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[151] Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. Adnostic: Privacy preserving targeted advertising. In *Network and Distributed System Security (NDSS) Symposium*, 2010.

[152] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in Numbers: Making Strong Anonymity Scale. In *OSDI*, 2012.

[153] Haiquan Zhao, Ashwin Lall, Mitsunori Ogihara, and Jun Xu. Global iceberg detection over distributed data streams. In *ICDE*, 2010.

[154] Qi (George) Zhao, Mitsunori Ogihara, Haixun Wang, and Jun (Jim) Xu. Finding Global Icebergs over Distributed Data Sets. In *Principles of Database Systems*, 2006.

İstemi Ekin Akkuş

https://www.mpi-sws.org/~iakkus

Interests

Computer Networks, Distributed Systems, Privacy

Education

Max Planck Institute for Software Systems,
Kaiserslautern and Saarbruecken, Germany **Sep 2010 - Sep 2015**
Doctoral Student, Computer Science

- *Max Planck Institute Scholarship recipient*
- Advisor: Professor Paul Francis

University of Toronto, Toronto, Ontario, Canada **Sep 2007 - Sep 2009**
Master of Applied Science, Electrical and Computer Engineering

- *University of Toronto Fellowship recipient*
- Thesis Title: Data Recovery for Web Applications
- Advisor: Professor Ashvin Goel
- Overall GPA: 3.90/4.0

Koç University, İstanbul, Turkey **Sep 2005 - Aug 2007**
Master of Science, Electrical and Computer Engineering

- *Merit Scholarship recipient*
- Thesis Title: Peer-to-peer Multipoint Video Conferencing Using Layered Video
- Advisors: Professor M. Reha Civanlar and Professor Öznur Özkasap
- Overall GPA: 3.96/4.0; 1st in class

Koç University, İstanbul, Turkey **Sep 2000 - Jun 2005**
Bachelor of Science, Mechanical Engineering
Bachelor of Science, Computer Engineering

- *Merit Scholarship recipient*
- *Double Major; graduated with Honors*

İstanbul (Erkek) Lisesi, İstanbul, Turkey **1992 - 2000**

- Language of Instruction: German
- Overall GPA: 4.85/5.0

Research Experience

Max Planck Institute for Software Systems, Kaiserslautern, Germany
Research Assistant **Sep 2010 - Sep 2015**

- Worked on design and implementation of large-scale distributed systems focusing on privacy.
- Dissertation Title: Towards a Non-tracking Web

International Computer Science Institute (ICSI), Berkeley, CA
Visiting Researcher **March 2014 - July 2014**

- Designed and developed Priv3+, a Firefox add-on that enables users to control the amount of tracking by any third-party (including social widgets and behavioral targeting cookies) with a general cookie policy without requiring a pre-defined blacklist. Supervised by Dr. Nicholas Weaver.

Microsoft Research India, Bangalore, India

Research Intern **July 2012 - Sep 2012**

- Worked on the design of a privacy-preserving, non-cloud digital assistant/ recommendation system to infer user intent regarding email content and present users with suggestions. Supervised by Dr. Saikat Guha.

University of Toronto, Toronto, Ontario Canada

Research Assistant **Sep 2007 - Sep 2009**

- Designed and developed a generic data recovery system for web applications. Advised by Prof. Ashvin Goel.
- Responsible for maintaining over 100 server machines.

Teaching Assistant **Sep 2008 - May 2009**

- Head TA for Computer Security and Operating Systems courses.

Koç University, Graduate School of Sciences and Engineering, İstanbul, Turkey

Research Assistant **Sep 2005 - Aug 2007**

- Worked on peer-to-peer multipoint video conferencing. Advised by Prof. M. Reha Civanlar and Prof. Öznur Özkasap.

Teaching Assistant **Sep 2005 - Aug 2007**

- Computer Networks, Operating Systems and Distributed Computing Systems courses.

Koç University, Rahmi Koç College of Engineering, İstanbul, Turkey

Undergraduate Researcher **Feb 2001 - Jul 2005**

- Conducted research on various multicast protocols using Network Simulator (ns-2). Supervised by Prof. Öznur Özkasap. (Feb 2003 - Jul 2005)

Publications

Journal

1. **Istemi Ekin Akkus**, Öznur Özkasap, M. Reha Civanlar. Peer-to-peer Multipoint Video Conferencing with Layered Video. *Journal of Network and Computer Applications*, Volume 34, Issue 1, January 2011, pp. 137-150.

International Conferences & Workshops

1. **Istemi Ekin Akkus**, Nicholas Weaver. The Case for a General and Interaction-based Third-party Cookie Policy. *IEEE Workshop on Web 2.0 Security and Privacy, W2SP2015*, San Jose, CA, USA, May 21, 2015.

2. Ruichuan Chen, **Istemi Ekin Akkus**, Paul Francis. SplitX: High-performance Private Analytics. *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM13*, Hong Kong, China, August 12-16, 2013.

3. **Istemi Ekin Akkus**, Ruichuan Chen, Michaela Hardt, Paul Francis, Johannes Gehrke. Non-tracking Web Analytics. *Proceedings of the 19th ACM Conference on Computer and Communications Security, CCS'12*, Raleigh, NC, USA, October 16-18, 2012.

4. Pramod Bhatotia, Alexander Wieder, **Istemi Ekin Akkus**, Rodrigo Rodrigues, Umut A. Acar. Large-scale Incremental Data Processing with Change Propagation. *3rd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'11*, Portland, OR, USA, June 14-15, 2011.

5. **Istemi Ekin Akkus**, Ashvin Goel. Data Recovery for Web Applications. *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks, DSN2010*, Chicago, IL, USA, June 28-July 1, 2010.

6. **Istemi Ekin Akkus**, Öznur Ozkasap, M. Reha Civanlar. Multi-objective Optimization for Peer-to-Peer Multipoint Video Conferencing Using Layered Video. *Proceedings of the 16th International Packet Video Workshop, PV2007*, Lausanne, Switzerland, November 12-13 2007.

7. **Istemi Ekin Akkus**, M. Reha Civanlar, Öznur Özkasap. Peer-to-peer Multipoint Video Conferencing Using Layered Video. *Proceedings of the IEEE International Conference on Image Processing, ICIP2006*, Atlanta, GA, USA, October 8-11 2006.

8. **Istemi Ekin Akkus**, Öznur Özkasap, M. Reha Civanlar. Secure Transmission of Video on an End System Multicast using Public Key Cryptography. *International Workshop on Multimedia, Representation and Security, MRCS2006*, İstanbul, Turkey, Sep 11-13 2006. Lecture Notes in Computer Science, vol. 4105, pp. 603-610, Springer-Verlag Heidelberg.

Technical Reports

1. **Istemi Ekin Akkus**. Formal Model and Verification of Privacy-preserving String Discovery for Mobile and Web Analytics. *Technical Report MPI-SWS-2014-006*, 2014.

2. **Istemi Ekin Akkus**, Ruichuan Chen, Paul Francis. String Discovery for Private Analytics. *Technical Report MPI-SWS-2013-006*, 2013.

Skills

Languages: Turkish (native), English (fluent), German (good), Italian (beginner)
Programming: C, Java, JavaScript, PHP, Python, UNIX Shell scripting, SQL
Operating Systems: UNIX, Linux, Mac OS X, MS Windows, DOS

Honors & Awards

- Max Planck Institute scholarship recipient (2010 - 2015)
- University of Toronto fellowship recipient (2007 - 2009)
- Vehbi Koç Foundation - Grant for Graduate study abroad (2007)
- The Scientific and Technological Research Council of Turkey (TUBITAK) Scholarship to support M.S. Thesis (2005 - 2007)
- Vehbi Koç Scholarship for graduate study, Koç University, covering tuition and monthly stipend (2005 - 2007)
- Vehbi Koç Scholarship for undergraduate study, Koç University, covering tuition, dormitory expenses and monthly stipend (2000 - 2005)
- Vehbi Koç Scholar High Honors List (once), Dean's Honor List (3 times), Koç University, (2000 - 2005)
- İstanbul Erkek Liseliler Vakfı, Alumni Foundation Scholarship for success in High School (1995 - 2000)

Professional Activities

- IEEE Student member
- ACM Student member

Extra-curricular Activities

- Two times intramural tournament champion (Basketball, Coed Div 2) at the University of Toronto (Fall 2007, Winter 2008)
- Organized charity events at Koç University for people with disabilities and children with leukemia (2003, 2004)
- Organization committee of Koç University Drama Festival (2003, 2004)
- Organization committee of Koç University International Debate Tournament (2002, 2003)

Personal

- Hobbies: Photography, rock climbing, running, cycling, basketball

References

Available upon request.