Security and Privacy of Hash-Based Software Applications

Amrit Kumar

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Amrit Kumar

Thèse dirigée par **Pascal Lafourcade**
et codirigée par **Cédric Lauradoux**

préparée au sein **d'Équipe Privatics, Inria, Grenoble-Rhône Alpes**
et de **l'École Doctorale MSTII**

Security and Privacy of Hash-Based Software Applications

Thèse soutenue publiquement le **20 octobre, 2016**,
devant le jury composé de :

Mr. Refik Molva
Professeur, Eurecom, Président
Mr. Gildas Avoine
Professeur, INSA Rennes, Rapporteur
Mr. Sébastien Gambs
Professeur, Université du Québec à Montréal, Rapporteur
Mr. Kasper B. Rasmussen
Professeur associé, University of Oxford, Examinateur
Ms. Reihaneh Safavi-Naini
Professeur, University of Calgary, Examinatrice
Mr. Pascal Lafourcade
Maître de Conférence, Université d'Auvergne, Directeur de thèse
Mr. Cédric Lauradoux
Chargé de Recherche, Inria, Co-Directeur de thèse

Acknowledgments

Firstly, I would like to express my sincere gratitude to my advisors Dr. Cédric Lauradoux and Dr. Pascal Lafourcade for their continuous support during my PhD study and for their patience and motivation. The research reported in this dissertation would not have been possible without their guidance and help. Besides research, I have found their advice and guidance about other issues in life to be invaluable as well. I could not have imagined having better and friendly advisors to complete this journey.

I would like to thank Prof. Gildas Avoine and Prof. Sébastien Gambs for reviewing this dissertation. I also extend my thanks to the rest of my PhD committee: Prof. Refik Molva, Prof. Reihaneh Safavi-Naini and Prof. Kasper B. Rasmussen, for accepting to be a part of the jury.

I am extremely grateful for the financial support provided by the Labex PERSYVAL-LAB (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir and also to MITACS for funding me an internship at University of Calgary. My sincere thanks goes to Prof. Reihaneh Safavi-Naini, who provided me an opportunity to join her team as an intern during my PhD, and for providing access to the research facilities during my stay at University of Calgary.

I thank my collaborators for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last three years. Last but not the least, I would like to thank my family: my parents and to my brother and sister and my fiancée for supporting me spiritually throughout my PhD and my life in general.

Abstract

Hashing and hash-based data structures are ubiquitous. Apart from their role in the design of efficient algorithms, they particularly form the core to many sensitive software applications. Whether it be in authentication on the Internet, integrity/identification of files, payment using Bitcoins, web proxies, or anti-viruses, the use of hashing algorithms might only be internal but yet very pervasive.

This dissertation studies the pitfalls of employing hashing and hash-based data structures in software applications, with a focus on their security and privacy implications. The mainstay of this dissertation is the security and privacy analysis of software solutions built atop *Bloom filters* — a popular hash-based data structure, and *Safe Browsing* — a malicious website detection tool developed by Google that uses hash functions. The software solutions studied in this dissertation have billions of clients, which include software developers and end users.

For Bloom filters and their privacy, we study a novel use case, where they form an essential tool to privately query leaked databases of personal data. While for security, we study Bloom filters in adversarial settings. The study encompasses both theory and practice. From a theoretical standpoint, we define adversary models that capture the different access privileges of an adversary on Bloom filters. We put the theory into practice by identifying several security related software solutions (employing Bloom filters) that are vulnerable to our attacks. This includes: a web crawler, a web proxy, a malware filter, forensic tools and an intrusion detection system. Our attacks are similar to traditional denial-of-service attacks capable of bringing the concerned infrastructures to knees.

As for Safe Browsing, we study vulnerabilities in the architecture that an adversary can exploit. We show several attacks that can simultaneously increase traffic towards both the Safe Browsing server and the client. Our attacks are highly feasible as they essentially require inverting hash digests of 32 bits. We also study the privacy achieved by the service by analyzing the possibility of re-identifying websites visited by a client. Our analysis and experimental results show that Safe Browsing can potentially be used as a tool to track specific classes of individuals.

This dissertation highlights the misunderstandings related to the use of hashing and hash-based data structures in a security and privacy context. These misunderstandings are the geneses of several malpractices that include the use of insecure hash functions, digest truncation among others. Motivated by our findings, we further explore several countermeasures to mitigate the ensuing security and privacy risks.

Keywords: Hashing, Bloom filters, Safe Browsing, Security, Privacy, Denial-of-service.

Résumé

Les fonctions de hachage et les structures de données qui leur sont associées sont omniprésentes dans le monde numérique. En dehors de leurs rôles dans la conception d'algorithmes efficaces, elles sont aussi essentielles dans le domaine de sécurité et de protection de la vie privée. Elles permettent de protéger les mots de passe stockés sur les serveurs d'authentification ou protéger l'intégrité lors d'échange de fichiers. Elles sont omniprésentes dans tous les protocoles cryptographiques comme ceux de signature électronique.

L'objective de cette thèse est d'analyser la sécurité et la protection de la vie privée des logiciels basés sur des fonctions de hachage. Dans ce contexte, nous nous intéressons plus particulièrement à deux applications. La première concerne les logiciels basés sur des filtres de Bloom — une structure de données basée sur le hachage. La deuxième est le service de "Safe Browsing". Ce service, développé par Google, fournit une méthode sécurisée pour naviguer sur Internet tout en détectant les sites malveillants (sites de *phishing* ou *malwares*). Les deux applications sont très populaires, ayant des milliards d'utilisateurs.

Pour des filtres de Bloom, nous étudions un nouveau cas d'utilisation, où ils constituent un outil essentiel pour interroger une base de données en respectant la vie privée. Nous analysons aussi la sécurité associée aux utilisations des filtres de Bloom dans un logiciel sensible. Notre étude aborde aussi bien les aspects théoriques que pratique. D'un point de vue théorique, nous définissons des modèles d'adversaire qui captent les différents accès au filtre pour un adversaire. Nous mettons la théorie en pratique en identifiant la vulnérabilité de plusieurs logiciels basés sur des filtres de Bloom. Cela comprend: un robot web, un proxy, un filtre de malware, les outils de forensique et un système de détection d'intrusion. Nos attaques ressemblent aux attaques par déni de service.

En ce qui concerne le service de "Safe Browsing", nous étudions les vulnérabilités dans l'architecture qu'un adversaire puisse exploiter. Nous montrons plusieurs attaques qui peuvent simultanément augmenter le trafic vers le serveur et le client. Nous étudions également le niveau de la vie privée assuré par le service en analysant la possibilité de réindentifier des visites aux certains sites web. Notre analyse et les résultats expérimentaux montrent que le service de "Safe Browsing" pourrait être potentiellement utile comme un outil pour tracer certaines classes des individus sur Internet.

Cette thèse met en lumière les idées fausses liées à l'utilisation de hachage et les structures de données basées sur le hachage dans le contexte de sécurité et de protection de la vie privée. Motivés par nos résultats, nous explorons en outre plusieurs contremesures pour augmenter le niveau de sécurité et de protection de la vie privée.

Mots-clés: Hachage, Filtres de Bloom, Safe Browsing, Sécurité, Vie privée, Déni de service

Contents

Acknowledgments iii
Abstract v
Résumé vii

1 Introduction **1**
1.1 Motivation 1
1.2 Problematics and Adopted Methodology 3
1.3 Contributions and Outline 5
1.3.1 Hash Functions for Privacy 5
1.3.2 Security and Privacy of Bloom Filters 5
1.3.3 Security and Privacy Analysis of Safe Browsing 6
1.4 Publications 6

2 Hashing and Hash-Based Data Structures in Security and Privacy **9**
2.1 Hash Functions 9
2.2 Hash Tables 11
2.3 Merkle Trees 13
2.4 Bloom Filters 15
2.4.1 Description 15
2.4.2 False Positive Analysis 16
2.5 Bloom Filter Variants 19
2.5.1 Counting Bloom Filters 19
2.5.2 Scalable Bloom Filters 20
2.6 Count-Min Sketches 21
2.7 Golomb-Compressed Sequences 23
2.8 Summary 25

I Security and Privacy Analysis of Hashing and Bloom Filters **27**

3 Pitfalls of Hashing for Privacy **29**
3.1 Introduction 29
3.2 Balls-into-Bins 31
3.2.1 Number of Empty Bins 31
3.2.2 Coupon Collector Problem 32
3.2.3 Birthday Paradox 32

3.2.4 Number of Bins with Load Greater than One 33
3.2.5 Maximum, Minimum and Average Load 33
3.3 Settings and Adversaries 35
3.3.1 Settings 35
3.3.2 Adversaries 36
3.4 One-to-One Mapping 37
3.5 Many-to-One Mapping 37
3.6 Gravatar Email Re-identification 38
3.6.1 Description 39
3.6.2 Re-identification 40
3.7 MAC Address Anonymization 41
3.7.1 One-to-One Anonymization 42
3.7.2 Many-to-One Anonymization 44
3.8 Summary 46

4 Privacy Provisions of Bloom Filters 47
4.1 How not to use a Bloom Filter 48
4.1.1 Naive Approach 48
4.1.2 Auditing Without Full Hashes: Blackhash 48
4.2 Bloom Filters for Better Privacy 49
4.2.1 Differential Privacy 50
4.2.2 BLIP 50
4.2.3 RAPPOR 52
4.3 Bloom Filters for Private Membership Queries 53
4.3.1 Context and Problem Statement 53
4.3.2 A Survey of Alerting Websites 54
4.4 Privacy-Friendly Alerting Websites 57
4.5 Solutions for Private Databases 58
4.6 Solutions for Public Databases 59
4.6.1 Private Information Retrieval 59
4.6.2 Private Membership Query Using PIR 60
4.6.3 Extension with PBR Protocol 62
4.7 Practicality of the Solutions 63
4.7.1 Applicability of PIR 63
4.7.2 Experimental Analysis 64
4.8 Related Work 66
4.9 Summary 67

5 Security of Bloom Filters 69
5.1 Introduction 70
5.2 Adversary Models 71
5.2.1 Chosen-insertion Adversary 73
5.2.2 Query-only Adversary 75

5.2.3 Deletion Adversary 77
5.2.4 Summary 77
5.3 Application Level Tools 78
5.3.1 Web Spider: Scrapy 79
5.3.2 Bitly Spam Filter: Dablooms 82
5.4 Networking Tools 83
5.4.1 Web Proxy: Squid 83
5.4.2 Intrusion Detection System: AIEngine 84
5.5 Forensic Tools 85
5.5.1 NSRL Forensic Filter 85
5.5.2 Extension to sdhash 86
5.6 Predicting Unknown Filter Parameters 87
5.6.1 Number of Hash functions 87
5.6.2 Filter Size 88
5.6.3 Hash Functions 89
5.7 Countermeasures 90
5.7.1 Alternate Data Structures 90
5.7.2 Worst-case Parameters for Bloom Filters 94
5.7.3 Probabilistic Solutions 95
5.8 Related Work 97
5.8.1 Attacks 97
5.8.2 Defense 98
5.9 Summary 100

II Security and Privacy of Safe Browsing 101

6 Safe Browsing 103
6.1 Online Phishing and Malware Threats 103
6.2 Overview of Safe Browsing Services 105
6.3 Google Safe Browsing 108
6.3.1 Lookup API 110
6.3.2 Safe Browsing API v3 112
6.3.3 Local Data Structures 115
6.3.4 Safe Browsing Cookie 117
6.3.5 Lookup API versus Safe Browsing API 118
6.4 Facilitating Safe Browsing Usage 118
6.4.1 Safe Browsing Diagnostic Tool 119
6.4.2 Reporting Tools 119
6.5 Effectiveness and Usability Studies 121
6.6 Yandex Safe Browsing 121
6.7 Summary 122

7 Privacy Analysis of Safe Browsing 125

7.1 Introduction 125
7.2 Single Prefix Hit 128
7.2.1 Analysis 129
7.3 Multiple Prefix Hits 130
7.3.1 Collisions on Two Prefixes 131
7.3.2 URL Re-identification 132
7.3.3 Statistics on Decompositions 134
7.3.4 A Tracking System based on Safe Browsing 137
7.4 Blacklist Analysis 139
7.4.1 Inverting Digests 139
7.4.2 Orphan Prefixes 140
7.4.3 Presence of Multiple Prefixes 141
7.5 Mitigations 142
7.6 Summary 143

8 (In)Security of Safe Browsing 145
8.1 Introduction 145
8.2 Related Work 146
8.3 Attacks on Safe Browsing 147
8.3.1 Threat Model 147
8.3.2 Attack Routine 148
8.3.3 False Positive Flooding Attacks 149
8.3.4 "Boomerang" Attacks 150
8.3.5 Impact 151
8.4 Feasibility of Our Attacks 152
8.4.1 Generating Domain Names and URLs 152
8.4.2 Generating Deadly Domains 153
8.4.3 Comparing the Domain Topologies 155
8.5 Countermeasures 156
8.5.1 Lengthening the Prefixes 156
8.5.2 Probabilistic Solutions 157
8.6 Ethical Considerations on Attack Demonstration 158
8.7 Responsible Disclosure and Impact 159
8.8 Summary 162

9 Conclusions and Perspectives 163

III Back Matter 169

List of Figures 171

List of Tables 175

Bibliography 177

Chapter 1

Introduction

Contents

1.1 Motivation **1**

1.2 Problematics and Adopted Methodology **3**

1.3 Contributions and Outline **5**

1.3.1 Hash Functions for Privacy 5

1.3.2 Security and Privacy of Bloom Filters 5

1.3.3 Security and Privacy Analysis of Safe Browsing 6

1.4 Publications **6**

1.1 Motivation

Hashing plays an important role in algorithms and data structures. In the most simple terms, hashing (or a *hash function*) can be thought of as a way to rename an address space. This renaming is particularly interesting as it provides a simple mean to compress the address space. In more concrete terms, hashing is a function that takes an input of arbitrary or almost arbitrary length and returns an output whose length is a fixed value also known as a *hash* or a *digest*.

Hashing is ubiquitous. To start with, hash functions efficiently solve some basic problems in Computer Science, such as searching (*e.g.*, Hash tables [CLRS09]), membership testing (*e.g.*, Bloom filters [Blo70]), statistics on streaming data (*e.g.*, Count-Min sketches [CM05]), *etc.* All these data structures are remarkably simple, a property that they owe to the underlying hash function. Even more importantly, most of these hash-based data structures are probabilistic and significantly improve upon their deterministic counterparts in terms of complexity (memory or time).

Additionally, hash functions also form a very useful tool for specific fields such as computer security and cryptography. With the advent of distributed computing in modern Internet applications, they have become increasingly important. Whether it be in authentication on the Internet using passwords, integrity/identification of files, payment

using Bitcoins, web proxies, or anti-viruses, the use of hashing algorithms might be internal but yet very pervasive. In general, if storage and lookup is an issue, systems often inadvertently fall back on hashing.

Interestingly, hashing was also one of the first methods used to implement the concept of *Proof-of-Work* system [DN93, Bac02], which provides a computational deterrence for *denial-of-service* (DoS) attacks and other service abuses such as spams by requiring some work from the service requester, usually meaning processing time by a computer. *Hashcash* [Bac02] is the simplest example of a proof-of-work system based on hashing. Bitcoin [Nak11] is a popular crypto-currency and is a very intricate example where a Hashcash-like proof-of-work is used. Hashing is also extensively used in privacy-preserving cryptographic protocols, such as *Private Set Intersection Protocols* [DCT12, DCGT12].

Hash-based data structures are equally popular within the security and privacy community. Among all, hash tables, Bloom filters (and their variants) [Blo70, FCAB00, ABPH07], and hash-based sketches such as count-min sketches [CM05] have found the most number of applications. Hash tables and Bloom filters are often used in situations were a service's ultimate overhead is storage and lookup. For instance, they are particularly a very popular choice in designing web proxies [Wes04], DNS cache [Ber01], web crawlers [ON10], web servers [LM10] and intrusion detection systems [AIE16, Pax98] among many others. Bloom filters alone have found numerous applications in networking: resource routing [BM05], web caching [FCAB00] or packet filtering [BCMR02].

Hash-based data structures have also been used in privacy-preserving cryptography and data privacy. As for privacy-preserving cryptography, they form an essential building block of efficient primitives such as searchable encryption [Goh03] and Private Set Intersection [DCW13] among others. While for data privacy, in particular data anonymization, hash-based data structures are fast becoming a popular choice to achieve *Differential Privacy* [Dwo06] — a strong notion of privacy for statistical databases. The most popular data structures employed in the domain of anonymization are Bloom filters and count-min sketches. Bloom filters for instance have been used in BLIP [AGK12, AGMT15] and Google's crowd sourcing statistics tool RAPPOR [EPK14] — currently incorporated in its Safe Browsing feature. A count-min sketch has been used to design practical techniques for privately gathering statistics from large data streams in [MDC15]. A practical application scenario is to instantiate a real-world privacy-friendly statistics tool for Tor[1] hidden services.

The afore-cited applications clearly reflect the extent of ubiquity of hashing and hash-based data structures in software applications designed to ensure security and privacy. This ubiquity and the extent to which it can impact services and end users form the motivation to study the security and privacy implications of employing hashing and hash-based data structures in software applications.

[1] https://www.torproject.org/

1.2 Problematics and Adopted Methodology

On the one hand, the usage of hashing in most of the afore-mentioned applications have been extensively studied and best practices have been well agreed upon and generally respected (such as in the case of digital signatures and password-based authentication). While, on the other hand, there exist several application scenarios of hashing and hash-based data structures where the role of hashing and its correct usage are largely misunderstood and sometimes even deliberately ignored. Typical examples include reducing the size of the output returned by a hash function, employing insecure hash functions to boost performance, or the security implications of certain hash-based data structures (for instance, Bloom filters and Hash tables) in sensitive software applications. The consequence of the lack of understanding can be catastrophic as many of these applications may drastically fail to ensure the required security and privacy guarantee. To this end, we discuss two well studied examples to highlight the impact of the usage of hashing and hash-based data structures for security and privacy. The first example is a bad yet popular practice in the field of privacy while the other relates to the usage of hash tables in a security context.

Let us consider the simplest usage of hashing to ensure privacy. In fact, the goal here is to achieve what is referred to in the literature as "*anonymity*". The functional use case is to take a US Social Security Number (SSN) and render it "anonymous", *i.e.*, render the SSN in a state from which it cannot be learned. The simplest method that one could apply is to hash the SSN using a suitable hash function that would not allow an attacker to recover the actual SSN from the output. Such functions do exist and are called secure hash functions. In face with a secure hashing mechanism, an adversary wishing to break the anonymity may attempt to run the hash function backward, *i.e.*, to find its inverse. Clearly, this inversion is bound to fail. A closer look to the SSNs however reveals the following fact: an SSN is a 9-digit number, therefore there are 10^9, *i.e.*, a billion SSNs in total. A clever attacker may hence hash all of the possible SSNs in advance, and build an index that allowed him to recover the SSN from its corresponding hash value in the blink of an eye. Hashing the SSN would offer no protection at all against such an attacker. Building this index would only require hashing a billion SSNs, which is highly feasible with the existing computing infrastructures. This example shows that hashing does not necessarily render data anonymous and hence it should be employed with caution. In fact, there are more advanced uses of hashing that can offer some protection in some settings. But the casual assumption that hashing is sufficient to anonymize data is risky at best, and usually wrong.

Our next example stems from the usage of hash tables in security related applications. We note that hash tables in average require a constant time for lookup. However, in the worst case, they may take a linear time. This renders a system employing hash tables prone to DoS attacks, where the goal of the attacker would be to force the system to systematically exhibit the worst-case behavior. This would eventually require the system to overuse the computing resources. An attacker can mount such attacks by inserting and querying hash tables with specially crafted inputs that force the hash table to fall in the worst-case scenario.

Many natural follow-up questions arise. For our example on SSN, an immediate question could be: What happens if instead of plain hashing, one also employs truncation, *i.e.*, reduce the size of the output of the hash function? Indeed, since digests are truncated, several SSNs may generate the same truncated digest and provide uncertainty in determining the actual SSN. As for our example on hash tables, a natural question would be to understand whether similar attacks exist when the underlying hash table is replaced by another data structure. Or, if employing a replacement data structure introduces new vulnerabilities. To this end, one may also ask, which is the best data structure to resist such attacks and what would be the ensuing trade-off between security/privacy and robustness.

This dissertation attempts to answer some of these questions by analysis hashing and its applications to security and privacy. The analysis in general follows a methodology that consists of four standard layered procedures described below:

1. **Identify vulnerabilities:** Analyze the security and privacy implications of employing hashing and hash-based data structures in certain sensitive software applications. The software applications considered here are deemed sensitive as their aim is to ensure an acceptable guarantee on the security and privacy of users depending on them. The goal would be to understand whether and under which conditions hash-based software applications are vulnerable to attacks.

2. **Build and test exploits:** An immediate objective would be to understand if these vulnerabilities that create security and/or privacy loopholes can be exploited to cause unintended or unanticipated behavior. From a security perspective, this behavior may include gaining control over the software system or allowing DoS attacks. While, in a privacy context, the goal would be to show a privacy breach caused due to the underlying vulnerability.

3. **Design theoretical frameworks:** While the previous two goals are practice oriented. The results obtained however would lead to designing theoretical framework (wherever possible) to study the attacks. To this end, we aim to study several adversary models that would capture different access privileges to the software application and their ensuing threat. Such a framework should be independent or to the very least should not depend much on the application in question and hence should be generic enough to study software applications of certain classes such as those using a particular data structure or employing a particular malpractice.

4. **Build secure solutions:** Once the vulnerabilities are identified, our final objective is to build secure solutions to ensure the minimum security and privacy guarantee required from the concerned applications. To this end, we aim to explore legacy-compatible countermeasures, as well as clean-slate solutions. These solutions should take into account the existing deployment or service constraints such as robustness of the system and memory constraints, *etc.*

1.3 Contributions and Outline

The contributions presented in this manuscript can be grouped into 3 broad segments. In the first segment presented in Chapter 3, we describe the pitfalls of hashing for data anonymization. In the second segment starting from Chapter 4, we discuss the security and privacy implications of employing Bloom filters. To begin with, in Chapter 4, we discuss how Bloom filters can contribute to the design of a privacy-preserving cryptographic protocol. In Chapter 5, we extend our behavioral study of Bloom filters in adversarial environments. The goal is to provide a security analysis of Bloom filters and in particular, several of its sensitive software applications. The last segment, starting from Chapter 6 is devoted to the security and privacy analysis of Safe Browsing services — a service provided by Google and other vendors to protect users from malicious websites. In the following, we provide a brief presentation of our contributions in these three segments.

1.3.1 Hash Functions for Privacy [CDKL16]

Boosted by strict legislations, data anonymization is fast becoming a norm rather than an exception. However, as of now no generic solution has been found to safely release data. As a consequence, companies often resort to ad-hoc means to anonymize datasets. Both past and current practices indicate that hashing is often thought to be an effective way to anonymize data. Unfortunately, this is rarely the case. We contribute to the systematization of knowledge to explain why hashing often fails. The argument follows the probabilistic model of *balls-into-bins* and applies it on two real-world case studies to illustrate how hashing fails to ensure the claimed privacy guarantee for data anonymization.

1.3.2 Security and Privacy of Bloom Filters [GKL15, KLL16, KL15]

We study whether Bloom filters are correctly designed/implemented in a security context. To this end, we construct adversary models for Bloom filters and illustrate attacks on six applications, namely Scrapy web spider, bitly dablooms spam filter, Squid cache proxy, forensic tools: NSRL Filter and `sdhash`, and an intrusion detection system called AIEngine. As a general impact of our attacks, filters are forced to systematically exhibit worst-case behavior. We explore several countermeasures such as computing the worst-case parameters in adversarial settings, strengthening Bloom filters without changing the parameters, and using an alternate data structure which resists better to our attacks. Most of these contributions were developed in [GKL15]. An extension of this work with stronger adversary models is developed in [KLL16], where we include attack scenarios in which the filter parameters are unknown to the adversary.

We also explore a potential use of Bloom filters in the context of privacy-preserving cryptography. We consider the problem of data leakage and what we call *alerting websites*. Alerting websites such as `haveibeenpwned.com` let users verify if their accounts have been compromised due to the leakage. We expose the privacy risks associated to 17 such websites, and evaluate existing solutions towards designing privacy-friendly alerting websites. Among other solutions that include *private set intersection*, we combine Bloom

filters and *private information retrieval* protocols for private membership testing. We also investigate the practicality of these solutions with respect to real world database leakages. These contributions are developed in [KL15].

1.3.3 Security and Privacy Analysis of Safe Browsing [GKL14, GKL16]

Safe Browsing is a very popular phishing and malware filter developed by Google and later also offered by Yandex. We analyze Google and Yandex Safe Browsing services from two perspectives: security and privacy. In order to analyze the privacy aspects, we quantify the privacy achieved by these services by analyzing the possibility of re-identifying URLs visited by a client. We thereby challenge Google's privacy policy which claims that Google cannot recover URLs visited by its users. Our analysis and experimental results show that Google and Yandex Safe Browsing can potentially be used as a tool to track specific classes of individuals. This privacy analysis of Safe Browsing services is developed in [GKL16].

We also study the Google Safe Browsing architecture through a security point of view. We propose several denial-of-service attacks that increase the traffic between Safe Browsing servers and their clients. Our attacks leverage the false positive probability of the data representation deployed to store the blacklists on the client's side and the possibility for an adversary to generate (second) pre-images for a 32-bit digest. These contributions on the security aspect of Google Safe Browsing are developed in [GKL14].

1.4 Publications

All of the work in this dissertation has been published or is in the process of being published in academic formats and venues.

Conference Publications

Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. A Privacy Analysis of Google and Yandex Safe Browsing. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 25-July 1*, 2016

Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. The Power of Evil Choices in Bloom Filters. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25*, 2015

Amrit Kumar and Cédric Lauradoux. A Survey of Alerting Websites: Risks and Solutions. In *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28*, 2015

Amrit Kumar, Pascal Lafourcade, and Cédric Lauradoux. Short Paper: Performances of cryptographic accumulators. In *39th IEEE Conference on Local Computer Networks, LCN 2014, Edmonton, AB, Canada, 8-11 September, 2014*, 2014

Amrit Kumar and Cédric Lauradoux. Private Password Auditing - Short Paper. In *Technology and Practice of Passwords - International Conference on Passwords, PASSWORDS'14, Trondheim, Norway, December 8-10, 2014, Revised Selected Papers*, 2014

Under Submission

Amrit Kumar, Cédric Lauradoux, and Pascal Lafourcade. Bloom Filters in Adversarial Settings, 2016. Submitted to ACM Transactions on Privacy and Security (TOPS)

Mathieu Cunche, Levent Demir, Amrit Kumar, and Cédric Lauradoux. The Pitfalls of Hashing for Privacy, 2016. Submitted to IEEE Communications Surveys & Tutorials

Others

Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. (Un)Safe Browsing. Research Report RR-8594, INRIA, September 2014

Chapter 2

Hashing and Hash-Based Data Structures in Security and Privacy

Contents

2.1 Hash Functions **9**
2.2 Hash Tables **11**
2.3 Merkle Trees **13**
2.4 Bloom Filters **15**
2.4.1 Description 15
2.4.2 False Positive Analysis 16
2.5 Bloom Filter Variants **19**
2.5.1 Counting Bloom Filters 19
2.5.2 Scalable Bloom Filters 20
2.6 Count-Min Sketches **21**
2.7 Golomb-Compressed Sequences **23**
2.8 Summary **25**

This chapter presents an overview of hash functions, and hash-based data structures that are commonly used in the domain of security and privacy. The data structures presented in this chapter are *Hash tables* [CLRS09], *Merkle trees* [Mer88], *Bloom filters* [Blo70], *Counting Bloom filters* [FCAB00], *Scalable Bloom filters* [ABPH07], *Count-Min sketches* [CM05] and *Golomb-Compressed Sequences* [PSS10]. For each data structure, we present its workings and discuss some of its applications from the domain of security and privacy. In addition to presenting the essential material, another goal of this chapter is to establish the notations that will be used in the rest of this dissertation.

2.1 Hash Functions

A hash function compresses inputs of arbitrary length to a digest/hash of fixed size. Mathematically speaking, a hash function usually denoted by h is of the following form $h : \{0,1\}^* \rightarrow \{0,1\}^\ell$, where ℓ is the digest size. Hash functions are normally assumed to

be *uniform*, which means that given a random input from $\{0,1\}^*$, each digest in $\{0,1\}^\ell$ is equally likely.

Hash functions are a very popular primitive used in algorithms [CLRS09] and in cryptography/security [MVO96]. The design of a "*good hash function*" however depends on the field of its application. From the context of their applications, hash functions can be broadly classified into two groups: *non-cryptographic* and *cryptographic* hash functions.

Non-cryptographic hash functions such as MurmurHash [App10] or Jenkins hash [Jen96] are designed to be fast, to uniformly distribute their outputs and to satisfy several other criteria such as the *avalanche test* [App10]. The avalanche test captures the phenomenon where, a slight change in an input (for example, flipping a single bit) propagates in a way that the output changes significantly (*e.g.*, half the output bits get flipped). The SMHASHER suite [App10] provides a good overview of these functions and the tests they must satisfy.

The design of a cryptographic hash function on the other hand is very different. Cryptographic hash functions are slower than their non-cryptographic siblings. The reason being that they are designed to be secure against known attacks. In order to defend against known attacks, a cryptographic hash function h must verify the following properties:

- **Pre-image resistance:** Given a digest d, it should be computationally infeasible to find an input x, such that $h(x) = d$.
- **Second pre-image resistance:** Given an input x and the digest $h(x)$, it should be computationally infeasible to find another input $x' \neq x$, such that $h(x) = h(x')$.
- **Collision resistance:** It should be computationally infeasible to find two inputs $x \neq x'$ such that $h(x) = h(x')$.

Let us consider a hash function h with ℓ-bit digests. The choice of ℓ is critical for cryptographic hash functions because the basic complexities for finding pre-images, second pre-images and collisions are 2^ℓ, 2^ℓ and $2^{\ell/2}$ respectively. The first two complexities correspond to brute force attacks — the best generic attack that an attacker can mount. The complexity of finding a collision is a consequence of the *birthday paradox* (see Chapter 3 for further detail). A cryptographic hash function is considered secure if there is no attack with a lower complexity. As non-cryptographic hash functions are generally designed with efficiency considerations, they do not satisfy these properties [CW03, AB12].

Because of the sensitive use cases of cryptographic hash functions, only a small set of cryptographic hash functions are in effective use today. The NIST recommendation for cryptographic hash functions are SHA-256, SHA-384, SHA-512 and SHA-3 [Nat12, Nat14].

In the following, we present two notions specific to this dissertation: *multiple (second) pre-images* and *digest truncation*. *Multiple pre-images* consider the case where, given a digest d, an attacker wishes to compute n pre-images x_i such that $h(x_i) = d$. Accordingly, we define *multiple second pre-images*, where given x' and $h(x')$, the attacker wishes to compute n second pre-images x_i such that $h(x_i) = h(x')$. Clearly, pre-image resistance is a priori required for multiple pre-image resistance.

Another notion which is of particular interest in the context of this dissertation is what we refer to as *digest trucation*. In fact, very often, applications need to reduce (or truncate) the size of the full digest generated by a hash function. The resulting digest, referred to as a *truncated digest* must be carefully used as explained in [Dan12], since it is commonly assumed that for a truncated digest of ℓ' bits the security is reduced at least to $2^{\ell'}$ (pre-image and second pre-image) and $2^{\ell'/2}$ (collision). If ℓ' is too small, computation of pre-images, second pre-images or collisions become feasible, and the initial security guarantee can no longer be maintained. For instance, in the case of GOOGLE Safe Browsing (see Chapter 6 for further details), SHA-256 is applied to URLs, but the 256-bit digests are truncated to 32 bits. With 32-bit digests, pre-images and second pre-images can be computed after 2^{32} brute force computations. This should roughly take between 1-3 hours on any commodity hardware.

Applications. Hash functions are crucial to security. The most elementary yet vital use of cryptographic hash functions is in the field of password-based authentication, where a password is used to authenticate a legitimate user of a service. In a password-based authentication, a password is presented by a user to the service provider, which is then compared with the one that was used to register on the service. On a secure server, the passwords are not stored in clear, but instead hashed, and the comparison is done using digests instead of passwords. A hashed password reduces the chance that an insider or an attacker who has the database of hashed passwords can recover a password — a consequence of its pre-image resistance. Due to its collision resistance, a cryptographic hash function ensures that no two users with different passwords will have the same hashed password, hence one cannot authenticate as the other.

Hash functions also form an essential tool in different cryptographic primitives and protocols. This includes digital signatures, message authentication codes, checking integrity of files, cryptographic accumulators (see [MVO96, KLL14] and references therein), proof-of-work as in Bitcoin, *etc.* Recently, hash functions have also become a bad yet popular choice to anonymize data sets. We saw in Chapter 1 an example of its usage to anonymize SSNs. More such privacy applications of hash functions are analyzed in Chapter 3.

2.2 Hash Tables

A hash table [CLRS09] is a data structure used for implementing the so-called *associative arrays*. Associative arrays are data structures that are similar to arrays but are not indexed by integers, but by other forms of data such as strings. In fact, arrays can be seen as a mapping, associating with every integer in a given interval some data item. However, there are many situations, where we want to index items differently than just by integers. Common examples are strings (for dictionaries, phone books, menus, database records), or structs (for dates, or names together with other identifying information). Hash tables are data structures to store such associative arrays, associating a *value* to a *key*. A typical example would be a phone book, where the key is the name of a person and the value is his/her telephone number.

A hash table can be represented as an array A of size m. To map a key to an entry of the array (*aka* a *bucket*), we first map a key to an integer and then use the integer to index the array. The first mapping is done using a hash function (usually non-cryptographic). Hence, given a key k, our access could then simply be $A[h(k) \bmod m]$, as taking modulo m reduces the digest to an integer less than m and hence yields an index of the array. The value corresponding to the key can then be stored at this index of the array. In order to query for a key, one repeats the same procedure to obtain an index of the array and look for the value stored. Ideally, each different key should be attributed to a different index, but in practice this is not achievable as collisions may occur. The most usual way to deal with collisions in hash tables is to store a linked list [CLRS09] at each index of the array. Such a list is also known as a *hash chain* in the context of hash tables. We note that there are other methods to handle collisions, including *open addressing* [CLRS09], wherein instead of using hash chains, the system follows a deterministic strategy to probe for an empty index, where the value can be inserted.

Figure 2.1 depicts a hash table representing a phone book, where collisions are handled by maintaining a linked list.

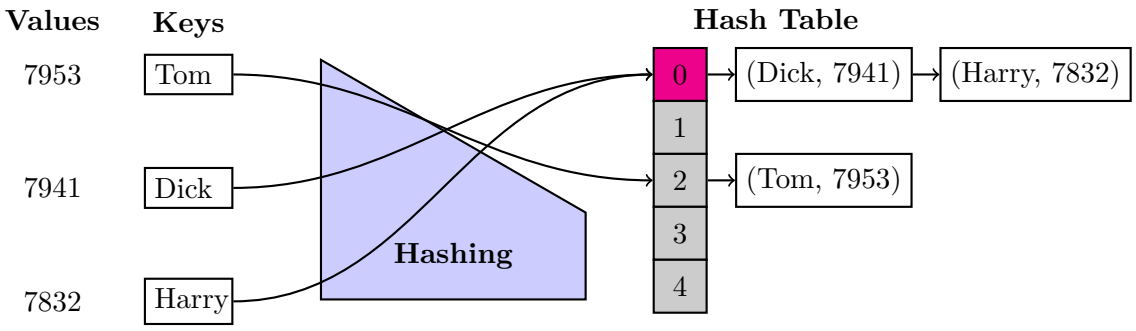

Figure 2.1: A hash table to represent a phone book. It contains three key-value pairs: ('Tom', '7953'), ('Dick', '7941') and ('Harry', '7832'). The colored cell represents a collision of the hash function on the keys 'Dick' and 'Harry'. A linked list is used to handle collisions. Keys are also stored along with the values to remove ambiguity for queries such as: What is the phone number of Dick?

Complexity. The average time to search, and therefore to insert or remove, an item in a hash table is $\mathcal{O}(1)$, which corresponds to the cost of computing a hash function. In the worst case, when all the items in the hash table are inside the same bucket, the complexity is linear in the size of the linked list, *i.e.*, $\mathcal{O}(n)$.

Applications. In many applications, associative arrays are implemented as hash tables because of their performance characteristics. This makes hash tables so common that they are primitive in many languages such as PHP, Python, Perl and Java. Hash tables are also commonly used to implement many types of in-memory tables, such as *caches*, which are auxiliary data tables used to speed up the access to data that is primarily stored in a slower media. In a similar vein, Hash tables have been employed in network proxies such as in SQUID [Wes04], where the proxy reduces the time to retrieve a resource from a

distant network entity. Another use case is Apache Cassandra [LM10], which is essentially a distributed hash table. Cassandra provides a distributed storage system for managing very large amounts of structured data spread out across many commodity servers, while providing highly available service with no single point of failure. Hash tables are also popular as a data structure to detect duplicate data or previously seen data. For instance, the `gzip` compression routine [Deu96] implements a hash table to detect duplicate strings in a text. Similar uses can be found in web crawlers which have to maintain a database of already visited web pages. Web applications such as the Apache web server parse and also collect fields from HTTP POST requests into hash tables automatically, so that they can be accessed by application developers.

2.3 Merkle Trees

A *hash tree* or a Merkle tree [Mer88] due to Merkle is a tree data structure useful to check the integrity of large data blocks of a file or a set of files. Currently the main use of a Merkle tree is to make sure that data blocks received from other peers in a peer-to-peer network are received undamaged and unaltered, and even to check that the other peers do not lie and send fake blocks.

A Merkle tree is essentially a tree of hashes corresponding to a set of data blocks. It is built in a bottom-up fashion. First, the leaf-nodes are built, which consists in hashing each of the data blocks. The non-leaf nodes in the tree are then built by hashing the digests of their respective children. Figure 2.2 depicts a simple Merkle tree for four data blocks. First, the data blocks are hashed to generate the leaves containing Hash-00, Hash-01, Hash-10 and Hash-11. At the next stage, one generates Hash-0, which is the result of hashing the concatenation of Hash-00 and Hash-01. At the highest level, the Root Hash denotes the top-level hash, *i.e.*, the hash of the concatenation of Hash-0 and Hash-1. We note that most Merkle tree implementations are binary (two child nodes under each node) but they can just as well use many more child nodes under each node.

A Merkle tree can be used to check the integrity of a set of data blocks in the following manner. First, a trusted source builds the Merkle tree for the data blocks. Then, any entity wishing to obtain the data blocks first acquires the Root Hash of the Merkle tree from this trusted source. Once the Root Hash is obtained, the Merkle tree received from any non-trusted peer can be checked for validity. This is done by checking the received hash tree against the trusted Root Hash, and if the hash tree is damaged or fake, the transfer of the actual data blocks can be rejected.

We note that the integrity of each branch of the tree can also be checked independently, even when the whole tree is not available. For example, in the example of Figure 2.2, the integrity of the second data block can be verified immediately if the tree already contains Hash-00 and Hash-1 by hashing the data block and iteratively combining the result with Hash-00 and then Hash-1 and finally comparing the result with the Root Hash. Similarly, the integrity of the third data block can be verified if the tree already has Hash-11 and Hash-1. This can be an advantage since it is efficient to split files up in very small data blocks so that only small blocks have to be re-downloaded if they get damaged.

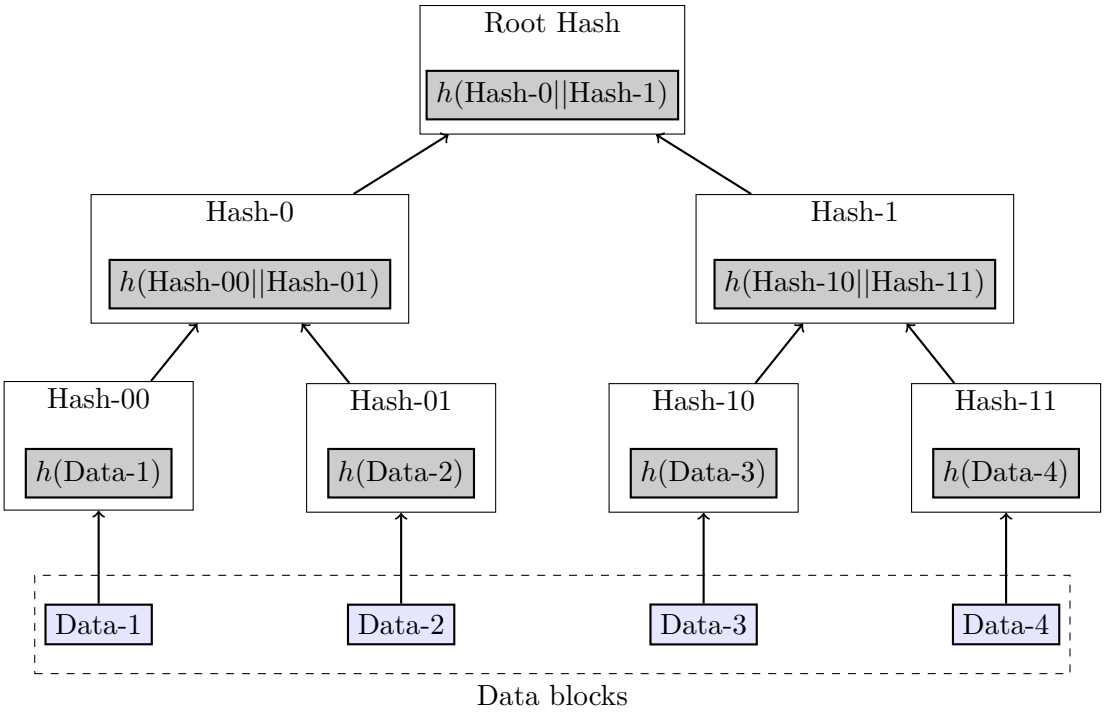

Figure 2.2: A binary Merkle tree with 4 data blocks as leaves. Hash-00 = h(Data-1); Hash-01 = h(Data-2); Hash-0 = h(Hash-00||Hash-01) = $h(h$(Data-1)||h(Data-2)), where || is the usual concatenation. Other nodes in the tree are defined in a similar manner.

Complexity. Building a Merkle tree requires $\mathcal{O}(2^n)$ hash computations, where n is the number of data blocks. As the trees has 2^n nodes, storing it tree requires $\mathcal{O}(2^n)$ space. Matching two Root Hashes to check the integrity of data blocks takes $\mathcal{O}(1)$ time. We note that demonstrating that a leaf node is a part of the given hash tree requires processing an amount of data proportional to the logarithm of the number of nodes of the tree. If the number of data blocks is large, then such a hash tree can become fairly big. However, if one small branch of the tree can be downloaded quickly, the integrity of that branch can be checked, and then the downloading of data blocks can start.

Applications. Merkle trees were initially conceived to efficiently handle one-time signatures [Mer88]. However, now they are being essentially used to verify any kind of data stored, handled and transfered in and between peers. As Merkle trees can efficiently check the integrity of large data exchanged between peers, they are ideal for peer-to-peer networks such as BitTorrent[1], Bitcoins and the Etherum network [KM16] that extends the Bitcoin protocol to build other protocols such as a cypto-contract protocol. Google Wave protocol[2] also uses Merkle trees. This protocol is under development and has the goal to provide near real-time communication between cooperative work *wave* servers (wave is an extension of emails). Merkle trees are also used in the IPFS and ZFS file systems[3], and Git distributed revision control system[4] among many others.

[1] http://www.bittorrent.com/
[2] http://www.waveprotocol.org/
[3] https://blogs.oracle.com/bonwick/entry/zfs_end_to_end_data
[4] https://git-scm.com/

2.4 Bloom Filters

Bloom filters introduced by Bloom [Blo70] is a space-efficient probabilistic data structure that provides an algorithmic solution to the *set-membership query problem*, which consists in determining if a given item belongs to a predefined set. To this end, Bloom filters offer a succinct representation of a set of items which can dramatically reduce space, at the cost of introducing *false positives*. If false positives do not cause significant problems, then Bloom filters may provide improved performance of an application.

2.4.1 Description

Essentially, a Bloom filter is represented by a binary vector $\vec{z}$ of size m. In the following, we define the *support* of a vector $\vec{z}$ of size m, *i.e.*, $\vec{z} = (z_0, \ldots, z_{m-1})$ denoted by $\text{supp}(\vec{z})$ as the set of its non-zero coordinate indexes:

$$\text{supp}(\vec{z}) = \{i \in [0, m-1], z_i \neq 0\} \ .$$

We also denote by $w_H(\vec{z})$, the Hamming weight of the filter $\vec{z}$, *i.e.*, the number of 1s in the filter. We use x with eventual subscripts to denote items inserted in the filter, while y with eventual subscripts to denote items queried to the filter. The set of items that the filter represents is denoted by $\mathcal{S}$.

In case of a classical Bloom filter, the bit vector $\vec{z}$ is initialized to $\vec{0}$. The filter is then incrementally built by inserting items from $\mathcal{S}$. Each item is inserted by setting certain bits of the filter to 1. By checking if these bits are set to 1, one may verify the belonging of an item to the filter. A detailed description of the operations follows:

Insertion. An item $x \in \mathcal{S}$ is inserted into a Bloom filter by first feeding it to k independent hash functions $\{h_1, \ldots, h_k\}$ (supposed to be uniform) to retrieve k digests modulo m: $I_x = \{h_1(x) \bmod m, \ldots, h_k(x) \bmod m\}$. These reduced hashes give k bit positions of $\vec{z}$. Finally, insertion of x in the filter is achieved by setting the bits of $\vec{z}$ at these positions to 1. Setting a bit to 1 is done independently of the previously stored value.

Since all the operations on digests are truncated modulo m in Bloom filters, for simplicity we omit mod m in the rest of the discussion.

Query. To determine if an item y belongs to $\mathcal{S}$, one checks if y has been inserted into the Bloom filter $\vec{z}$. Achieving this requires y to be processed (as in insertion) by the same hash functions to obtain k indexes of the filter, $I_y = \{h_1(y), \ldots, h_k(y)\}$. If the bit of $\vec{z}$ at any of these indexes is 0, then the item is not in the filter, otherwise if $I_y \subseteq \text{supp}(\vec{z})$, the item is present (with a small false positive probability).

Example 2.4.1 (A sample Bloom filter) *Figure 2.3 presents a Bloom filter of size $m = 12$, constructed using $k = 2$ hash functions. The inserted set consists of 3 items, $\mathcal{S} = \{x_1, x_2, x_3\}$. A collision occurs on one of the reduced digests of x_1 and x_3 (colored cell). More precisely, $h_2(x_1) \bmod m = h_1(x_3) \bmod m = 4$. Items y_1, y_2, y_3 are checked for*

belonging. The item $y_3 \notin \mathcal{S}$ is found to be present in the filter and hence is a false positive. The item y_1 however does not belong to $\mathcal{S}$.

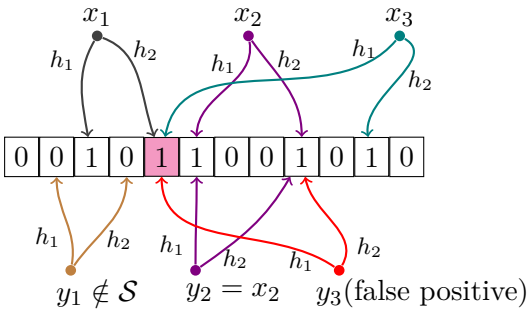

Figure 2.3: A Bloom filter with $m = 12$ and $k = 2$.

Complexity. Bloom filters can store items of arbitrary size using only m bits of space and the insertion/query runs in $\mathcal{O}(1)$ time: computation of k hash functions. However, this space and time efficiency of Bloom filters, comes at the cost of false positives.

2.4.2 False Positive Analysis

False positives arise due to collisions on reduced digests. We present below the rate with which they occur. The analysis follows the exposition given in [BM05].

Let n be the number of insertions into the filter. Then, the probability of a false positive for an item not in the initial set can be estimated in a straightforward manner given the assumption that hash functions are perfectly random. The probability that a certain bit is not set to 1 by a certain hash function is $\left(1 - \frac{1}{m}\right)$. The probability that this bit is not set by any of the k hash functions is $\left(1 - \frac{1}{m}\right)^k$. After n insertions, probability that the specific bit is still 0 is:

$$p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}} .$$

The approximation follows from the fact that for large m and small k, the following limit holds:

$$\lim_{x \to \infty} \left(1 - \frac{1}{x}\right)^{-x} = e .$$

As an extension, the probability that the specific bit is set to 1 is:

$$1 - \left(1 - \frac{1}{m}\right)^{kn} .$$

By definition, a false positive occurs when the bit corresponding to each of the k positions for an item is 1. Therefore, assuming that hash functions are independent, the probability of a false positive is:

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k .$$

For large m and relatively small k, we have:

$$f \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \quad . \tag{2.1}$$

For instance, the Bloom filter of Figure 2.3 with $n = 3$, $k = 2$ and $m = 12$ has a false positive probability of 0.15, *i.e.*, 15 out of every 100 random items will be a false positive. We note that (2.1) is not the most accurate approximation of the false positive probability (see [BGK+08] for a more accurate result). However, it is often used in software implementations and hence we abide by it throughout this dissertation.

There are two competing forces behind the false positive probability. On the one hand, using more hash functions gives a higher chance to find a bit not set for an item which is not a member of the filter, while on the other hand, using fewer hash functions increases the fraction of bits not set in the filter and hence decreases the false positive probability. Hence, fixing m and n, we choose k to minimize the false positive probability. We let $f = exp(g)$ where $g = k \ln(1 - e^{-kn/m})$. Taking the derivative of g with respect to k, we have:

$$\frac{\partial g}{\partial k} = \ln(1 - e^{kn/m}) + \frac{kn}{m} \cdot \frac{e^{-kn/m}}{1 - e^{-kn/m}} \; .$$

The zero of the derivative is:

$$k = \frac{m}{n} \cdot \ln 2 \; .$$

and can be proved to be the global minimum of f.

Hence, the optimal number of hash functions that minimizes the false positive probability is:

$$k_{\text{opt}} = \frac{m}{n} \cdot \ln 2 \quad , \tag{2.2}$$

and the corresponding false positive probability satisfies:

$$\ln\left(f_{\text{opt}}\right) = -\frac{m}{n} \cdot (\ln 2)^2 \quad . \tag{2.3}$$

Another important result is on the expected number of set bits in the filter. Let X be the random variable representing the number of 0s in the filter. After the insertion of n random elements, it follows that:

$$\begin{aligned} \mathbb{E}(X) &= \sum_{i=0}^{m} i \cdot \mathbb{P}\left(X = i\right) \\ &= \sum_{i=0}^{m} i \cdot \binom{m}{i} p^i (1-p)^{m-i} \\ &= mp \quad , \end{aligned} \tag{2.4}$$

where, $p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}}$.

Hence, in the optimal case, the expected number of 0s in the filter is $\frac{m}{2}$. Using a simple martingale argument, Broder and Mitzenmacher [BM05] further show that the exact fraction of unset bits is extremely concentrated around its expectation. Specifically, they prove using the Azuma-Hoeffding inequality, that for any $\epsilon > 0$,

$$\mathbb{P}\left(|X - mp| \geq \epsilon m\right) \leq 2e^{-\epsilon^2 m^2/nk} \quad . \tag{2.5}$$

The previous equations on the false positive rate of Bloom filters are well-established. However, they are valid as long as the digests of the inputs are uniformly distributed. We demonstrate in Chapter 5, the effect of non-uniformly chosen inputs on the false positive probability.

Union and Intersection of Bloom filters. The simple structure of Bloom filters makes certain operations very easy to implement. For example, let us consider two Bloom filters of the same size and built using the same hash functions. If these filters represent two sets $\mathcal{S}_1$ and $\mathcal{S}_2$, then, the Bloom filter that represents the union of the two sets can be obtained by taking the OR of the two bit vectors of the Bloom filters.

Bloom filters can also be used to approximate the size of the set represented by a Bloom filter. This result can also be extended to approximate the size of the intersection and the union of two sets.

Let us first consider a Bloom filter A of size m, built using k hash functions. Then, according to the result of Swamidass and Baldi [SB07], the number of items that was inserted into the filter can be approximated using the following formula:

$$n_A = -\frac{m \ln\left(1 - \frac{\bar{X}(A)}{m}\right)}{k},$$

where, $\bar{X}(A)$ is the number of 1s in the filter A.

Now, let us consider two Bloom filters A and B of equal size m and built using the same set of hash functions, then the approximate number of items in the union of the two filters (which somewhat represents the union of their corresponding sets) is given by:

$$n_{A \cup B} = -\frac{m \ln\left(1 - \frac{\bar{X}(A \cup B)}{m}\right)}{k},$$

where, $A \cup B$ on the right hand side represents the bitwise OR of A and B. Similarly, the size of the intersection of the two filters can be obtained by:

$$n_{A \cap B} = n_A + n_B - n_{A \cup B}.$$

Applications. Bloom filters have found many applications in the recent past. The earliest use of Bloom filters was to design a spell checker [Blo70]. However, more and more security and privacy-related software solutions have started to incorporate Bloom filters. Some of these sensitive software solutions include Web proxies [Wes04], web crawlers [Scr16], resource routing algorithms, intrusion detection systems [AIE16], forensic tools [Nat16, Rou10], *etc.* Interested readers may refer to [BM05] for more network related applications.

Bloom filters have also been used to design privacy-preserving tools. This notably includes BLIP [AGK12, AGMT15] and RAPPOR [EPK14]. In BLIP, the main objective

is to privately compute in a distributed manner the similarity between user profiles by relying only on their Bloom filter representations, while RAPPOR is a tool developed by Google to gather usage statistics. An experimental version of RAPPOR was also included in Google Chrome. These tools are presented in further detail in Chapter 4.

2.5 Bloom Filter Variants

It is worth noticing that a basic Bloom filter can only represent a static set. As a consequence, it neither allows for querying the multiplicities of an item, nor does it support the deletion of an item from the filter. In this section, we present two variants of Bloom filters which overcome these shortcomings: *Counting Bloom Filters* and *Scalable Bloom Filters*. The former solves the problem of multiplicities, while the latter can handle dynamic sets.

2.5.1 Counting Bloom Filters

A Counting Bloom filter (CBF) [FCAB00] allows to represent a multiset of items. In a CBF, the vector $\vec{z} = (z_0, \ldots, z_{m-1})$, represents a vector of w-bit counters, with $z_i \in \{0,1\}^w$. In fact, regular Bloom filters can be considered as CBFs with a counter size of one bit. Insertion of an item x now consists in incrementing the counter at the positions in I_x obtained by hashing. The query to retrieve the count of an item y involves computing the hash indices and obtain its set of counters C_y stored at these indices. The minimum value in the set C_y is then returned as the frequency estimate. CBFs also allow deletions. Deletions can be done by decrementing the relevant counters, however they may generate false negatives since other items may share the same counter.

There exist two main issues with CBFs. We discuss them one by one. First, the problem of *counter overflows*. A counter overflow occurs when a counter value reaches $2^w - 1$, and cannot be incremented anymore. In case of a counter overflow, one typically stops counting as opposed to overflowing and restarting at 0. However, this strategy introduces undercounts. In the following, we study the probability that a given counter overflows. Let $c(i)$ be the count corresponding to the i-th counter. We use the previous notation m to denote the size of a CBF, *i.e.*, a CBF is composed of m counters of size w. The probability that the i-th counter is increased j times after the insertion of all the n items can be estimated as follows:

$$\mathbb{P}(c(i) = j) = \binom{nk}{j} \left(\frac{1}{m}\right)^j \left(1 - \frac{1}{m}\right)^{nk-j} \leq \binom{nk}{j} \left(\frac{1}{m}\right)^j \leq \left(\frac{enk}{jm}\right)^j, \tag{2.6}$$

where, the last inequality is obtained from the well-known upper bound of the binomial coefficient:

$$\binom{n}{k} \leq \left(\frac{en}{k}\right)^k.$$

By substituting the optimal number of hash functions $k = \frac{m}{n} \cdot \ln 2$ in (2.6), we obtain:

$$P(c(i) = j) \leq \left(\frac{e \ln 2}{j}\right)^j. \tag{2.7}$$

The second problem related to the use of CBFs concerns the choice of the parameter w. A large w quickly diminishes the space savings obtained from using a Bloom filter. A small w on the other hand may quickly lead to the saturation of counter values. As such, choosing the right value is a difficult trade-off that depends on the distribution of the items in the multiset. However, the analysis from Fan *et al.* [FCAB00] shows that in typical use cases that appear in practice, $w = 4$ should be a reasonable choice to prevent counter overflows. This is because if $w = 4$, then $j = 16$ is a critical value for the overflow. Continuing with (2.7) and by using a union bound argument:

$$\mathbb{P}(c(i) \geq 16) \leq \sum_{j=16}^{\infty} \left(\frac{e \ln 2}{j}\right)^j \leq \sum_{j=16}^{\infty} \left(\frac{e \ln 2}{16}\right)^j \leq \sum_{j=16}^{\infty} \left(\frac{1}{8}\right)^j.$$

A simple computation of the sum of the geometric series gives:

$$\mathbb{P}(c(i) \geq 16) \leq \frac{8}{7} \times 2^{-48}.$$

Consequently, the probability that any counter amongst the m overflows is:

$$\mathbb{P}(\max_i c(i) \geq 16) \leq \frac{8}{7} \times 2^{-48} \times m,$$

which is extremely small for any m to be encountered in practice.

Despite the fact that a small value of w suffices in most settings, an obvious disadvantage of CBFs is that they appear quite wasteful of space. Using counters of four bits blows up the required space by a factor of four over a standard Bloom filter, even though most entries are zero.

2.5.2 Scalable Bloom Filters

A Scalable Bloom Filter (SBF) [ABPH07] addresses the problem of having to choose an a priori maximum number of items n to be inserted in the filter. SBFs can work with an arbitrary number of items while keeping the false positive probability reasonable at the cost of memory size.

An SBF is a collection of Bloom filters created dynamically. To each filter is associated an insertion counter. When the counter reaches a certain threshold δ, a new filter is created with a counter set to 0. When an item is queried to an SBF filter, it is searched for in all its constituent filters.

Let us denote f_i to be the false positive probability of the i-th filter for $i \geq 0$. At a given moment, the data structure consists of λ filters with error probabilities satisfying:

$$\forall\ 1 \leq i \leq \lambda - 1,\ f_i = f_0 r^i,\ \text{with } 0 < r < 1\ .$$

Then, the compounded false positive probability F for the SBF is:

$$F = 1 - \prod_{i=0}^{\lambda-1} (1 - f_i)\ .$$

The authors in [ABPH07] suggest to use an r in the interval $[0.8, 0.9]$ for better space usage.

Several other variants of Bloom filters exist in the literature such as Compressed Bloom Filters, Spectral Bloom Filters, Stable Bloom Filters. Interested readers may refer to [TRL12] for a survey of Bloom filters and their variants.

2.6 Count-Min Sketches

The Count-Min (CM) sketch introduced in [CM05] is a probabilistic data structure for summarizing data streams. A *data stream* consists of a continuous set of data that can be received by an application. An example application could be a network monitor receiving a constant flow of packets for traffic analysis.

CM sketches are somewhat similar to Bloom filters; the main distinction being that Bloom filters represent sets, while CM sketches represent multisets (as done by CBFs) and their frequency tables using a sub-linear space. In the following, we describe the *data stream model* and in the sequel, we present the CM data structure and a query procedure.

Data stream model. A *data stream* is represented by a vector $\vec{a}$, which is presented in an implicit, incremental fashion. The vector $\vec{a}$ is of dimension n, and its current state at a time t is given as: $\vec{a}(t) = [a_1(t), \ldots, a_i(t), \ldots, a_n(t)]$. Initially, $\vec{a}(0)$ is the zero vector $\vec{0}$. New data arrives in the form of updates. Each update is presented as a stream of pairs which modifies one entry of the vector. The t-th update is (i_t, c_t), where i_t is the index of the entry to be updated, while c_t is the value to be added to the entry. The t-th update (i_t, c_t) modifies the entries of the vector $\vec{a}(t)$ in the following manner:

$$a_{i_t}(t) = a_{i_t}(t-1) + c_t; \quad a_{i'}(t) = a_{i'}(t-1)\ \forall i' \neq i_t.$$

Hence, update to an entry changes its value by c_t, while other entries remain unchanged. At any time t, a *query* calls for computing certain functions of interest on $\vec{a}(t)$.

Data structure. A CM sketch hash two tunable parameters (ϵ, δ). These parameters measure the accuracy guarantees of the sketch, meaning that the error in answering a query is within a factor of ϵ with probability δ. The space and update time will consequently depend on these parameters.

The sketch is in fact represented by a two-dimensional array of size $d \times w$, where $d = \lceil \ln \frac{1}{\delta} \rceil$ and $w = \lceil \frac{e}{\epsilon} \rceil$. We denote this array by `count[1..d][1..w]`. Additionally, d hash functions $\{h_1, \ldots, h_d\}$, where $h_i : [1, n] \to [1, w]$ are chosen uniformly at random from a pairwise-independent family.

Update. The array `count[1..d][1..w]` is initialized to zero. For an update pair (i_t, c_t), the quantity c_t is added to one count in each row; the position is determined by h_j.

Formally, set $\forall 1 \leq j \leq d$:

$$\texttt{count}[j][h_j(i_t)] \leftarrow \texttt{count}[j][h_j(i_t)] + c_t.$$

Figure 2.4 depicts the update operation using 5 independent hash functions.

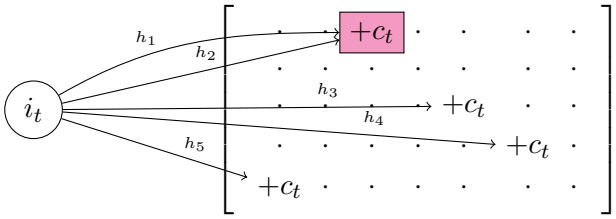

Figure 2.4: Updating the `count` matrix for a data stream using $d = 5$ hash functions and $n = 7$. The colored entries correspond to the collision of hash functions h_1 and h_2.

Query. The sketch is named after the two basic operations used to answer a *point query*, *i.e.*, a query to return an approximation of a_i. These operations are counting first and then computing the minimum. We explain below the details.

Let us consider point queries, denoted $\mathcal{Q}(i)$: Return an approximation of a_i. If for all updates $a_{i_t}(t) > 0$, then an estimated value $a_i^\dagger$ for the point query $\mathcal{Q}(i)$ is given by:

$$a_i^\dagger = \min_{1 \leq j \leq d} \texttt{count}[j][h_j(i)].$$

This requires finding the digests $h_1(i), \ldots, h_d(i)$; looking up in the matrix the values stored at $\{(1, h_1(i)), \ldots, (d, h_d(i))\}$ and taking the minimum of these values.

This estimate of a_i is bound to have an error, however it has been shown in [CM05] that the estimate $a_i^\dagger$ has the following guarantees: $a_i \leq a_i^\dagger$ and with probability at least $1 - \delta$, $a_i^\dagger \leq a_i + \epsilon \|\vec{a}\|_1$, where $\|\cdot\|_1$ is the ℓ_1 norm[5]. The first guarantee says that the CM sketch never under-estimates the value of a_i, while the second says that the probability that the error is within a factor ϵ is at most δ.

The sketch also allows other fundamental queries in data stream summarization such as computing the *range* (which consists in computing the sum of certain vector entries), and *inner product* queries of two vectors to be approximately answered very quickly. In addition, it can be applied to solve several other important problems in data streams such as finding quantiles. Interested readers may refer to [CM05] for details on how CM sketches answer to other types of queries.

Complexity. Time to update and produce the estimate is $\mathcal{O}(\ln \frac{1}{\delta})$, which corresponds to the computation of d hash functions and computing the minimum. The space used $\mathcal{O}(\frac{1}{\epsilon} \ln \frac{1}{\delta})$, which corresponds to the size of the `count` matrix.

[5] $\|\vec{a}\|_1 = \sum_{i=1}^{n} |a_i|$

Applications. CM sketches have found various applications in networking and databases. A count-min sketch is used for robust aggregation in sensor networks [KBC+05, NGSA08]. Another application of CM sketches is in traffic monitoring, accounting and network anomaly detection. The efficiency and tunable error rate is useful in scenarios where the network administrator needs to be able to detect high-volume traffic clusters in near real-time [ZSS+04, CJC10, RZ06, SVG10, BCCL07]. CM sketches and their extensions have also been used for several applications in data streaming and databases in general [PRUV10, SP06, AM04, HLCX11, GK04]. For instance, Pietracaprina *et al.* [PRUV10] use a CM sketch to build an approximation of the distribution of the frequencies of the items in a dataset. The innovative contribution is a method for building this approximation without querying the sketch for each item.

Recently, CM sketches have also been used to design privacy-preserving data mining tools. The most relevant solution is the one studied by Melis *et al.* in [MDC15]. Melis *et al.* design a practical technique for privately aggregating statistics from large data streams. In fact, the private aggregation is performed over CM sketches, rather than the raw inputs. This allows to reduce communication and computational complexity from linear to logarithmic in the size of the inputs. The resulting private statistics tool can be used to instantiate other protocols, and build systems addressing varied applications, such as a recommender system, or to detect hidden-Tor activities in a network. The latter allows Tor developers to collect traffic statistics such as the number of, and traffic generated by, hidden services, in order to fine tune design decisions and convince their funders of the value of the network [EDG14].

2.7 Golomb-Compressed Sequences

Golomb-Compressed Sequences [PSS10] (GCS) are a probabilistic data structure conceptually similar to Bloom filters, but with a more compact in-memory representation, and a slower query time.

A Bloom filter with an optimal number of hash functions occupies a space in memory that is $n \times \log_2(e) \times \log_2(1/f)$ bits. To put things into perspective, let us say one wants to store 100,000 elements with 1 false positive every 8,000 items. For these parameters, the optimal filter will be of size $100000 \times 1.44 \times 13$ bits $\approx$ 1870 Kb $\approx$ 1.83 Mb. The theoretical minimum for a similar probabilistic data structure has been shown to be $n \times \log_2(1/f)$ (see [CFG+78] for a proof). This gives a size of 1.2 Mb, so a Bloom filter roughly uses 44% more memory than the theoretically lower bound. GCS is in fact a way to get closer to that minimum. GCS is well suited for situations where one wants to minimize the memory occupation and can afford a slightly higher computation time, compared to a Bloom filter.

The key underlying idea of a GCS is to apply an encoding technique called *Golomb encoding* [Gol66] to a set of symbols from an alphabet (such as integers) whose frequencies decrease geometrically. Such a set can be obtained from another set of symbols where each symbol is equally likely (for instance, a set of hash values).

Let us first explain the encoding algorithm that applies to a set of integers with frequencies that decrease geometrically. We later explain how one may obtain this set from

a set of arbitrary strings.

The encoding of a set requires encoding each integer in the set one by one and the final encoding of the set is the concatenated encodings of each. We present below Algorithm 1 to encode an integer entry N of the set. The algorithm takes another parameter 2^p, where $1/2^p$ is the probability of false positives.

Algorithm 1: Golomb encoding of an integer N.

Data: Integer N and a false positive probability $1/2^p$.

$(q, r) \leftarrow$ `EuclideanDiv`$(N, 2^p)$;
`qc` $\leftarrow 1^q$; `// Quotient code`
`qc` $\leftarrow$ `qc`$||0$;
`rc` $\leftarrow$ `IntegerToBinary(r)` ; `// Remainder code, Length of rc is` p
return (`qc`$||$`rc`;)

Example 2.7.1 *If the false positive probability is* $\frac{1}{2^6}$*, i.e.,* $p = 6$*, for the integer* $N = 151 = 2 \cdot 2^6 + 23$*,* `qc` *is* 110 *corresponding to the quotient* 2 *and* `rc` *is* 010111 *corresponding to the remainder* 23*. The final encoding of* N *is* `qc` $\|$ `rc` $= 110010111$*.*

Decoding simply constitutes in reversing the encoding algorithm: q is set to be the number of 1s until the first zero and r is set to be the decimal representation of the remaining bit-sequence. N then can be reconstructed using q, r and 2^p.

Now, let us discuss how may one transform a set of arbitrary strings into a set of integers for which the probabilities decrease geometrically. First, each string is hashed which generates a set of hash values which are equally probable. At this stage, each message digest is mapped to an integer in $[0, n \times 2^p[$. The set of these integers is then sorted and the difference set of the sorted integers is computed. The entries of the difference set will now have probabilities that decrease geometrically (see [Gol66] for a proof). Golomb compression technique can hence be applied to the obtained set of values.

We note that to know if a value is in the initial set, a naive sequential decoding algorithm to decode each compressed value would incur linear cost. This is essentially because the compressed sequence does not allow random access. Hence, GCS [PSS10] augments the encoding algorithm with an index data structure which allows to seek to a position in the sequence which is close to the desired position. To this end, the range $[0, n \times 2^p[$ is divided into intervals of size T. In addition to this, a pointer to the beginning of the subsequence in each interval is stored in the index data structure. Since the hash values are uniformly distributed, each interval will contain roughly N/T values, so by seeking into it while querying, one only requires to decode only $1/T$-th of the whole GCS, thus getting a $T\times$ speed in query time. Clearly, this decreases the computational time by a constant factor, but does not affect the asymptotic complexity. Moreover, there is a trade-off between space and time: A smaller I reduces the query time but increases the size of the index data structure.

Complexity. The combined size of the encoded data is smaller than the size of the corresponding Bloom filter but remains $\mathcal{O}(n)$ and the asymptotic time complexity for a query is $\mathcal{O}(n)$.

Applications. Google Chromium, for instance, uses GCSs to keep a local (client) set of SSL certificate revocation list. The goal is to have a low memory occupation which is specifically important in memory-constrained scenarios (*e.g.*, mobile). In such scenarios, one can afford the data structure to be a little bit slower than Bloom filters since it is still much faster than an SSL handshake which entails a double network round-trip. Another use of Golomb encoding is in Google Safe Browsing version 4 API[6], where the server needs to send a list of hashes to the client. Since, usual compression algorithms cannot compress hashes (as they are equally probable), Golomb encoding offers a good compression alternative and reduces the required bandwidth.

2.8 Summary

This chapter presented an overview of hash functions and popular hash-based data structures from the field of security and privacy. It is clear that hash-based data structures are ubiquitous as they solve a wide variety of problems while being both memory and time efficient. In Table 2.1, we summarize the contents of this chapter.

Table 2.1: A summary of the data structures presented in this chapter. a in the time complexity for hash tables and GCS denotes the average case.

Data Struc.	**Purpose**	**Sample use case**	**Space**	**Time**
Hash tables	Associative arrays	Cache	$\mathcal{O}(n)$	$\mathcal{O}^a(1)$
Merkle trees	Data/file integrity	BitTorrent	$\mathcal{O}(2^n)$	$\mathcal{O}(1)$
CM sketch	Statistics on streams	Data mining	$\mathcal{O}(\frac{1}{\epsilon}\ln\frac{1}{\delta})$	$\mathcal{O}(\ln\frac{1}{\delta})$
GCS	Membership testing	Revocation list	$\mathcal{O}(n)$	$\mathcal{O}^a(n)$
Bloom filters	Membership testing	Cache, private matching	$\mathcal{O}(n)$	$\mathcal{O}(1)$

We note that Bloom filters will remain the focal point of Chapter 4 and Chapter 5. The former studies Bloom filters from a privacy perspective, while the latter studies them in a security setting. The latter study will also include its variants: Counting Bloom filters and Scalable Bloom filters. The reported work on Bloom filters can be seen as an extension of previous works on hash tables, which hence will be a point of comparison with the state of the art. Additionally, as count-min sketches can be seen as an extension of Bloom filters, our security study of Bloom filters trivially extend to count-min sketches. Golomb encoding will be revisited in Chapter 4 and Chapter 8, where it is used to compress digests corresponding to user identifiers and URLs.

[6] https://developers.google.com/safe-browsing/v4/

Part I

Security and Privacy Analysis of Hashing and Bloom Filters

Chapter 3

Pitfalls of Hashing for Privacy

Contents

3.1 Introduction **29**

3.2 Balls-into-Bins **31**

3.2.1 Number of Empty Bins 31

3.2.2 Coupon Collector Problem 32

3.2.3 Birthday Paradox 32

3.2.4 Number of Bins with Load Greater than One 33

3.2.5 Maximum, Minimum and Average Load 33

3.3 Settings and Adversaries **35**

3.3.1 Settings 35

3.3.2 Adversaries 36

3.4 One-to-One Mapping **37**

3.5 Many-to-One Mapping **37**

3.6 Gravatar Email Re-identification **38**

3.6.1 Description 39

3.6.2 Re-identification 40

3.7 MAC Address Anonymization **41**

3.7.1 One-to-One Anonymization 42

3.7.2 Many-to-One Anonymization 44

3.8 Summary **46**

3.1 Introduction

Personally identifiable information, or in general, data with social or economic value are collected by almost all web service providers. The data help a service provider identify, contact or locate discrete individuals, which in turn may be necessary to deliver the intended service. For instance, services like Google analytics[1] collect IP addresses (among

[1] www.google.com/analytics

other data) and generate valuable information such as the percentage of clients from a given city or those who use a specific web browser. Another example is that of *Wi-Fi tracking* [ME12], wherein individuals are tracked in the physical world for profiling or analytics purposes. These systems work by passively collecting unique identifiers such as MAC addresses emitted by Wi-Fi enabled portable devices.

The personally identifiable information handled by these service providers can however be used to the detriment of an individual's privacy. For instance, a location-based service provider may monitor an individual's activities during a day and may infer sensitive information such as his/her religion and sexual orientation from the points of interests he/she visits. Hence, much of the success of these services depends on the ability to ensure that the privacy of each individual is respected. One of the most effective ways to do so is *data anonymization*, sometimes also referred to as *data de-identification*. In simple terms, data anonymization consists in processing personally identifiable data in order to irreversibly prevent identification, hence, mitigating the privacy risks for the concerned individuals. Data anonymization can be applied either when the data is collected, communicated, or stored or when the data is published in the public domain for further analysis or scrutiny.

With privacy being an increasing concern, data anonymization is rapidly becoming a norm rather than an exception. This particularly owes to the different legislations and directives that have been enacted in the recent past [opi14, opi10, opi11, usj06, epr02]. In face with the obligation to anonymization, data custodians face a difficult choice between the possibility of *data re-identification* also known as *data de-anonymization* and the utility of the anonymized data. Considering this trade-off between utility and convenience, companies and administrators often turn to *pseudonymization* instead of full scale anonymization [Goo14a, PW07, Euc15, Goo15a]. Pseudonymization can be seen as a way to hide the real identity of an individual (or an identifier such as his social security number) by replacing it with a false one so that information pertaining to the individual can be handled (as usual) without knowing the real identity. It reduces the linkability of a dataset with the original identity of a data subject and hence mitigates privacy risks.

A popular choice to pseudonymize a data is through hashing [Goo14a, PW07, Euc15, Goo15a]. More precisely, cryptographic hash functions are often used to pseudonymize identifiers as these functions are *one way*, *i.e.*, pre-image resistant. This means that it is hard to obtain the identifier corresponding to a given *pseudonym* (the result of pseudonymization). This protects the identities of the individuals, but only ostensibly so. Indeed, while employing a hash function may appear to be a good security rationale, the final results of the "*hashing for anonymization*" approach has been catastrophic. In fact, re-identification attacks have been made possible on the published datasets and consequently sensitive data are now available in the wild [Bon13, Goo14a, aol06].

There are three main reasons for this failure. First, the *one-wayness* of hash functions is misunderstood and data custodians often underestimate the risk of exhaustive search. Second, even when exhaustive search cannot be carried out on the initial domain space, it should however be possible to do so on one of its subsets. This may allow an attacker to learn whether a user belongs to a given subset and hence eventually reveal a defining

property of the user. Finally, the following privacy argument is quite prevalent among data custodians and software developers: Hashing can not take into account any prior adversary knowledge. The argument clearly undermines the adversarial strength.

This chapter puts into light the difficulty to (pseudo)anonymize data with hash functions. It presents compelling evidences that highlight some of the mistakes made in the past when data custodians have followed this practice. The chapter is divided into three parts. The first part presents the probabilistic model of *balls-into-bins* which models anonymization using hashing. Some of the results presented in this part will also be used in later chapters. The second part establishes which privacy argument can be used when hash functions are used for anonymization. To this end, we employ the results on balls-into-bins to study the *anonymity set size* of a pseudonym. The final part focuses on real world case studies. We analyze mistakes made in two applications namely, Gravatar – a global avatar and Wi-Fi tracking systems. The study on Gravatar is due to Bongard [Bon13] that we formalize in this chapter. We extend the underlying idea in [Bon13] to analyze the case of Wi-Fi tracking systems. Later in this dissertation, we discuss another such application called *Safe Browsing*, where the privacy analysis becomes more involved and hence is treated separately in Chapter 7.

3.2 Balls-into-Bins

Balls-into-Bins is a classical setting in probability theory which models several interesting problems in Computer Science. The setting can be described in the following manner: We throw m balls into n bins sequentially by placing each ball into a bin chosen independently and uniformly at random. The setting leads to several natural problems, for instance, the probability that there is a bin with at least two balls in it, the maximum number of balls in any bin, the probability that none of the bins is empty, the average number of empty bins, *etc.* Solutions to these problems have found varied applications in different scientific fields. In this chapter, we apply the framework as a tool to analyze the privacy achieved when hashing is used to anonymize identifiers. To this end, we present in this section some well known results on balls-into-bins.

3.2.1 Number of Empty Bins

We note that since the balls are thrown uniformly at random, there is a possibility that some bins remain empty even after all the balls have been thrown. It is hence interesting to know the expected number of bins which will remain empty with reasonably high probability.

Let X be a random variable representing the number of empty bins, and X_i for $1 \leq i \leq n$ be random variables defined in the following way:

$$X_i = \begin{cases} 1, & \text{if bin } i \text{ is empty,} \\ 0 & \text{otherwise .} \end{cases}$$

Clearly, $X = \sum_{i=1}^{n} X_i$. Also, since balls are thrown independently and uniformly at random, the expected value of X_i is given by:

$$\mathbb{E}[X_i] = \mathbb{P}[\text{bin } i \text{ is empty}] = \prod_{j=1}^{n} (1 - \mathbb{P}(\text{ball } j \text{ goes into ball } i)) = \left(1 - \frac{1}{n}\right)^m \quad . \tag{3.1}$$

Equation 3.1 consequently yields the expected number of empty bins, which is given by,

$$\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \mathbb{P}[\text{bin } i \text{ is empty}] = n \cdot \left(1 - \frac{1}{n}\right)^m \quad .$$

One could also use the union bound to obtain the following probability:

$$\mathbb{P}[\exists \text{ empty bin}] \leq \sum_{i=1}^{n} \mathbb{P}[\text{ bin } i \text{ is empty}] \leq n \cdot \left(1 - \frac{1}{n}\right)^m \leq n \cdot e^{-\frac{m}{n}} \quad . \tag{3.2}$$

It should be noted that for $m \approx n$, we have $\mathbb{P}[\exists \text{ empty bin}] \leq \frac{n}{e}$. For large enough n, this gives a bound greater than 1, which is useless in this case. Therefore, in order that the bound be meaningful, we need to have $n \cdot e^{-\frac{m}{n}} \ll 1$, in which case $m \gg n \cdot \ln(n)$. However, in the latter case, the probability that there exists an empty bin is very small. For more discussions, interested readers may refer to [MR95].

3.2.2 Coupon Collector Problem

The *coupon collector problem* is defined as: for a given set of coupons of n types, how many times we have to choose the coupons (randomly and with replacement) to collect coupons of every type. This problem can be paraphrased as the problem of computing the average number of balls needed to have all bins non-empty.

Let X be the random variable that denotes the number of balls thrown into the bins needed to get at least one ball in each of the n bins. It is hence interesting to compute $\mathbb{E}[X]$ and $\text{Var}[X]$. It can be shown that (for details see [MR95]),

$$\mathbb{E}[X] = n \cdot (\ln(n) + o(1)) \,, \quad \text{and} \quad \text{Var}[X] = \frac{\pi^2}{6} \cdot n^2 + o(n^2) \quad . \tag{3.3}$$

Using Chebyshev's inequality, one can also show that the value of X is highly concentrated around its mean.

3.2.3 Birthday Paradox

Another related problem very often referred to in cryptography is the *birthday paradox*, which concerns the probability that, in a set of n randomly chosen people, some pair of them will have the same birthday. The problem leads to a well-known cryptographic attack called the *birthday attack*, which uses this probabilistic model to reduce the complexity of finding a collision for a hash function. In terms of balls and bins, the problem refers to computing the probability to have a bin with at least two balls into it.

We note that when $m > n$, by the pigeonhole principle, the probability that there exists a bin with at least two balls into it is 1. We hence restrict to a more interesting scenario where $m \leq n$. A collision is said to occur if some bin ends up containing at least two balls. We are interested in $C(m, n)$, the probability of a collision. One can show that:

$$1 - e^{-m(m-1)/2n} \leq C(m, n) \leq \frac{m(m-1)}{n} \quad . \tag{3.4}$$

Conversely, the number of balls required to generate a collision with probability p is given by:

$$m \approx \sqrt{2n \cdot \ln\left(\frac{1}{1-p}\right)} \quad .$$

These results are developed in further detail in [MVO96].

In the terminology of hash functions which generate ℓ-bit digests, the expected number of random samples that should be generated before getting a collision (with reasonably high probability) is not 2^ℓ, but rather only $2^{\ell/2}$ (roughly). This is obtained by substituting n by 2^ℓ in the previous equation.

3.2.4 Number of Bins with Load Greater than One

This problem is sometimes referred to as the *birthday problem* [MR95], where the goal is to find the number of days on which more than 1 person have the same birthday, *i.e.*, the number of bins with load greater than one.

One may obtain the required result by analyzing a random variable Y that represents the number of *conflicts*, that is, the number of pairs of balls that are in the same bin. It is easy to see that if we define $Y_{i,j}$ such that,

$$Y_{i,j} = \begin{cases} 1 & \text{balls } i \text{ and } j \text{ are in the same bin and } i \neq j, \\ 0 & \text{otherwise,} \end{cases}$$

then, $Y = \sum_{i=1}^{m} \sum_{j=i+1}^{m} Y_{i,j}$. Furthermore, the probability that two balls will be allocated in the same bin is exactly $\frac{1}{n}$. Therefore,

$$\mathbb{E}[Y] = \sum_{i=1}^{m} \sum_{j=i+1}^{m} \mathbb{E}[Y_{i,j}] = \binom{m}{2} \cdot \frac{1}{n} = \frac{m(m-1)}{2n}. \tag{3.5}$$

Equation 3.5 also implies that if the number of balls is greater than $\sqrt{n}$, then we expect to see at least one bin with more than one ball. This corresponds to the previous result on birthday paradox.

3.2.5 Maximum, Minimum and Average Load

As the title suggests, we group three results here, namely the maximum, minimum and the average load of a bin. These constitute the most relevant results in the context of anonymization using a hash function. Details are provided in the following sections.

Maximum load. The problems discussed in the previous section have been extensively studied in mathematics for many decades, while the problem of analyzing the maximum load has been studied only very recently, because of its wide scale applications in hashing, online load balancing, data collection, routing, *etc.*

Theorem 3.2.1 due to Raab and Steger [RS98] summarizes all results on maximum load. For instance, the theorem implies that, in the case when $m = n$, the maximum load of any bin is with high probability $\frac{\log n}{\log \log n}(1 + o(1))$. Furthermore, for the case $m \geq n \log n$, the maximum load of any bin is $\Theta\left(\frac{m}{n}\right)$. The case when $m \gg n$ is particularly important for the load-balancing scenario, where we have n servers accepting m job requests. It measures how the asymmetry between different servers grows over time when more and more requests arrive.

Theorem 3.2.1 (Raab and Steger [RS98]) *Let M be the random variable that counts the maximum number of balls into any bin. If we throw m balls independently and uniformly at random into n bins, then $\mathbb{P}[M > k_\alpha] = o(1)$ if $\alpha > 1$ and $\mathbb{P}[M > k_\alpha] = 1 - o(1)$ if $0 < \alpha < 1$, where:*

$$k_\alpha = \begin{cases} \frac{\log n}{\log \frac{n \log n}{m}} \left(1 + \alpha \frac{\log^{(2)}\left(\frac{n \log n}{m}\right)}{\log \frac{n \log n}{m}}\right), & \text{if } \frac{n}{\text{polylog}(n)} \leq m \ll n \log n, \\ (d_c - 1 - \alpha) \log n, & \text{if } m = c \cdot n \log n, \\ \frac{m}{n} + \alpha \sqrt{2 \frac{m}{n} \log n}, & \text{if } n \log n \ll m \leq n \cdot \text{polylog}(n), \\ \frac{m}{n} + \sqrt{\frac{2m \log n}{n}\left(1 - \frac{1}{\alpha} \frac{\log^{(2)} n}{2 \log n}\right)}, & \text{if } m \gg n \cdot (\log n)^3. \end{cases} \tag{3.6}$$

The constant term d_c depends on c and is a solution of $1 + x(\log c - \log x + 1) - c = 0$.

Minimum load. Let us first count the number of ways to throw m balls into n bins such that each bin has at least k balls into it. Since, each of the bins must have at least k balls, we can remove k balls from each bin. After removing these nk balls, we are left with the task of distributing $m - nk$ balls into n bins. The number of ways of distributing $(m - nk)$ balls into n bins is therefore:

$$\binom{m - nk + n - 1}{n - 1} \tag{3.7}$$

The above term is obtained using the *stars and bars* method from combinatorics due to Feller [Fel68]. Hence, the probability that the minimum load of a bin is *at least* k can be given as:

$$\frac{\binom{m-nk+n-1}{n-1}}{n^m} \tag{3.8}$$

Finally, the probability that the minimum load of a bin *is exactly* k is given by:

$$\frac{\binom{m-nk+n-1}{n-1} - \binom{m-nk-1}{n-1}}{n^m}, \tag{3.9}$$

where, the second binomial in the numerator counts the number of ways to throw m balls into n bins such that every bin has at least $k+1$ balls. This is obtained from (3.7) by substituting k by $k+1$.

The probability term given in (3.9) yields a generic formula to estimate the minimum load of a bin. Ercal-Ozkaya [EO08] proves the following specific result when $m > n \log n$.

Theorem 3.2.2 (Ercal-Ozkaya [EO08]) *For a constant $c > 1$, if one throws m balls into n bins where $m \geq cn \log n$ uniformly at random, then with high probability, the minimum number of balls in any bin is $\Theta\left(\frac{m}{n}\right)$.*

Average load. Let X_i be the random variable which counts the number of balls in the bin i. Clearly,

$$\sum_{i=1}^{n} X_i = m.$$

Therefore, $\mathbb{E}[\sum X_i] = m$ and by linearity of expectation, $\mathbb{E}[X_i] = \frac{m}{n}$. Hence, if $m = n$, we expect to see one ball in each bin.

3.3 Settings and Adversaries

In this section, we describe the settings and the notations when hash functions are used in the context of pseudonymization. We later define attacks and mount them to two real-world examples.

3.3.1 Settings

Let us assume that we want to create pseudonyms for a finite set $\mathcal{A}$ of m identifiers $\mathrm{id}_1, \ldots, \mathrm{id}_m$ using a one-way hash function of output length ℓ. We obtain a set $\mathcal{P}$ consisting of the following pseudonyms $h(\mathrm{id}_1), \ldots, h(\mathrm{id}_m)$ such than $|\mathcal{P}| \leq m$. We note that an inequality rather than an equality is used here because collisions can occur and hence pseudonyms may not necessarily be all distinct. For the rest of the discussion, we denote the size of $\mathcal{P}$ by n.

The notations m and n used here is in line with the notations used in the previous section on balls-into-bins, where m denotes the number of balls and n the number of bins. This is because identifiers represent the balls and the hash function generates pseudnoyms which represent the bins. The hash function ensures that given a random identifier, each pseudonym is equally likely. Hence, the hash function models the fact that each ball is thrown uniformly at random into the bins.

It is important to distinguish two scenarios depending on the number of identifiers associated to a pseudonym. In the first case, at most one identifier is associated to a pseudonym. This *one-to-one mapping* ($m = n$) implies that the hash function is injective. This case occurs when $m \ll 2^{\ell/2}$ (Birthday paradox bound). Figure 3.1 schematically presents this case.

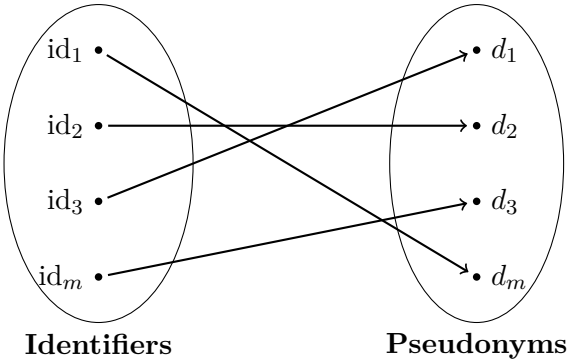

Figure 3.1: A one-to-one pseudonymization.

The second case considers the scenario where multiple identifiers get associated to the same pseudonym. It occurs when $m \gg 2^{\ell}$. This implies that the hash function is not injective, *i.e.*, the mapping is *many-to-one*. Figure 3.2 schematically presents this case.

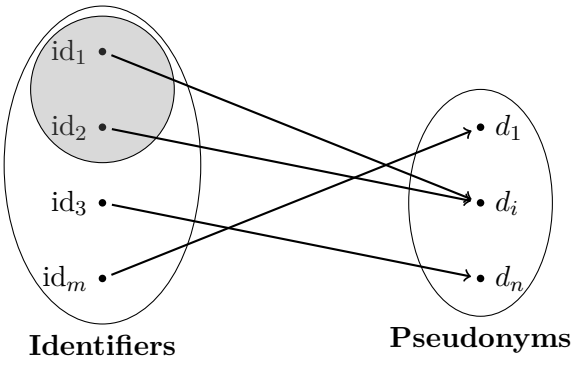

Figure 3.2: A many-to-one pseudonymization. id_1 and id_2 yield the same pseudonym d_i.

3.3.2 Adversaries

Pseudonyms so generated can be used in a database or in a protocol. From the knowledge of a pseudonym d, we need to establish the goals for an adversary. She can either achieve a *re-identification* of the identifier or attempt to mount a *discrimination* attack. These are the most common and standard attacks against any anonymization scheme (see [opi14]).

- The re-identification attack is the worst attack that may be mounted on an anonymized identifier. It implies that the adversary is able to invert the anonymization function.
- The discrimination attack requires the adversary to determine if a pseudonym belongs to a certain group or to another. The attack assumes that the domain $\mathcal{A}$ can be split into two: if an identifier verifies a certain discriminatory property it belongs to $\mathcal{A}_1$, otherwise it belongs to $\mathcal{A}_2$. Clearly, we have $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$ and $\mathcal{A}_1 \cap \mathcal{A}_2 = \emptyset$.

In the following sections, we study these adversaries under the two hypotheses on the underlying hash function.

3.4 One-to-One Mapping

The re-identification attack is possible in this case if m can be enumerated by the adversary in reasonable time. She can compute:

$$h(\mathrm{id}_1), \ldots, h(\mathrm{id}_m),$$

until she finds $h(\mathrm{id}_i) = d$. Since there is no collision, she recovers the unique identifier that corresponds to the pseudonym d. The complexity of this attack is $\mathcal{O}(m)$.

The discrimination attack is essentially a re-identification attack on an enumerable subset of identifiers. The adversary computes the pseudonyms $h(\mathrm{id}_i), \ldots, h(\mathrm{id}_j)$ associated to the subset $\mathcal{A}_1$ which is assumed to be enumerable in reasonable time. If any of the identifiers in $\mathcal{A}_1$ generates d as its pseudonym, then one can learn that d corresponds to an identifier that verifies the discriminating property associated to $\mathcal{A}_1$. Otherwise, d verifies the property associated to $\mathcal{A}_2$. In the former case, the adversary also learns the unique identifier, while in the latter case, she only learns that the identifier does not verify the discriminating property. The complexity of the discrimination attack is $\mathcal{O}(|\mathcal{A}_1|)$.

The afore-described attacks are feasible as long as the number of identifiers m is small compared to the number of possible digests, *i.e.*, we have a unique mapping between the pseudonyms and the identifiers. If m is small, an exhaustive search enables the re-identification attack. Otherwise, discrimination attacks are still possible even without any prior knowledge or auxiliary information. It clearly jeopardizes the use of hash functions to create pseudonyms. These issues are particularly detailed in Section 3.7.

3.5 Many-to-One Mapping

In this scenario, we consider a different privacy model in which several identifiers are associated to the same pseudonym. Hence, the risk of de-anonymizing an identifier is reduced. This model known as the *anonymity set* was discussed by Pfitzmann and Köhntopp [PK00]:

> *"Anonymity is the state of being not identifiable within a set of subjects, the anonymity set".*

Using a hash function h, a pseudonym d and for a given size r, the *anonymity set* of d is composed of r distinct identifiers $\mathrm{id}_1, \mathrm{id}_2, \ldots, \mathrm{id}_r$ that map to the same pseudonym d, *i.e.*,

$$h(\mathrm{id}_1) = h(\mathrm{id}_2) = \cdots = h(\mathrm{id}_r) = d. \tag{3.10}$$

Hence, d is anonymous among these r identifiers. In other words, the *anonymity set size* of d is r. The larger the value of r is, the better is the anonymity achieved.

For a pseudonym to have an anonymity set size of $r > 1$, one must have $m \gg 2^\ell$. This is unlikely to happen if we work with the full digest of a cryptographic hash function ($\ell \geq 256$). Hence, the digest size needs to be reduced to force multiple collisions. In the rest of this chapter, we assume without loss of generality that this is achieved by ignoring the most significant bits of the digest.

We note that when considering n distinct pseudonyms having anonymity sets of respective sizes $r_1, r_2, \ldots, r_n$, the overall anonymity achieved is given by the smallest size $\min_{1 \leq i \leq n} r_i$. This is in line with the argument that the anonymization technique for a set of identifiers gives only as much anonymity as its least anonymized identifier. The computation of this value and the required size of the reduced digests can be obtained using the balls-into-bins model of Section 3.2. In terms of ball-into-bins, this corresponds to computing the minimum load of a bin.

Now, we compute the average anonymity achieved for a set of n distinct anonymity sets of respective sizes $r_1, r_2, \ldots, r_n$, *i.e.*, compute the mean value of these r_is. Using the terminology of balls and bins, this is equivalent to computing the average load of a bin. As seen in Section 3.2, if $m = n$, we expect to see one ball in each bin. In terms of anonymity set size, this corresponds to an anonymity set size of 1, which implies that on an average there is no collision among the pseudonyms. This case degenerates into the case of one-to-one mapping. Hence, if the set of identifiers $\mathcal{A}$ were enumerable, then it should be possible to mount the re-identification attack. The discrimination attack may also succeed when $\mathcal{A}$ is not enumerable as long as a subset of it remains enumerable.

In some scenarios one may also be interested in knowing the maximum anonymity set size that can be achieved. When considering n distinct anonymity sets of respective sizes $r_1, r_2, \ldots, r_n$, the anonymity is the highest for the identifier id_i which has the maximum anonymity set size. This amounts to computing $\max_{1 \leq i \leq n} r_i$, which can be done using Theorem 3.2.1.

A critical hypothesis for the notion of anonymity set size is to assume that all the identifiers which form a given anonymity set have the same probability of appearance. If this condition does not hold, the anonymity set size can be reduced and re-identification or discrimination attacks become plausible. This criticism was made by Serjantov and Danezis in [SD02].

In the following sections, we apply the afore-described techniques to understand the level of anonymity achieved in two real-world applications. In Section 3.6, we consider the case of a large value of m in the context of email addresses. Despite a large m, this case in effect constitutes a one-to-one pseudonymization. In Section 3.7, we show how anonymization of MAC addresses fails as the domain space can be reduced to mount discrimination attacks in certain Wi-Fi tracking systems. This usecase allows to study both one-to-one and many-to-one mappings.

3.6 Gravatar Email Re-identification

In this section, we first present Gravatar, an avatar that can be recognized globally over different web services and in the sequel we present a de-anonymization attack proposed by Bongard [Bon13] to re-identify the email address of a user.

3.6.1 Description

Gravatar[2], a portmanteau of *globally recognized avatar*, is a service that allows a member of forums and blogs to automatically have the same profile picture on all participating sites (where the member is registered with the same email address). Several prominent web services such as GitHub, Stack Overflow and WordPress among others employ Gravatar. The service is available in the form of a plugin and as an API [PW07], and hence can be easily incorporated into any web service. Moreover, Gravatar support is provided natively in WordPress as of version 2.5 and in web-based project management application Redmine[3] with version 0.8 and later. Support for Gravatars is also provided via a third-party module in the Drupal[4] web content management system.

Gravatars work in the following way: users first create an account on gravatar.com using their email address, and upload an avatar to be associated with the account. They may optionally enter a variety of profile information to associate with their Gravatar account. This information together with the avatar is then openly-accessible by any participating web service. More precisely, whenever a user interacts with a participating service that requires an email address, such as writing a comment on Stack Overflow, the service checks whether that email address has an associated avatar on gravatar.com. If so, the Gravatar is shown along with the comment.

Gravatar requests. In order to retrieve the Gravatar of a user, the web service makes a request to gravatar.com. This requires no authentication, and is based around simple HTTP GET requests. The web service first generates the MD5 digest of the user's email address and then requests for the avatar using the URL: http://www.gravatar.com/avatar/digest, where 'digest' is the MD5 hash of the email address. See Figure 3.3 for a schematic representation. In case the Gravatar is required to be served over SSL, the URL for the request is modified to: https://secure.gravatar.com/avatar/digest. Gravatar also provides a number of built-in options which allow a developer to configure different parameters such as pixel, default image, *etc.* The profile of a user can also be accessed using a similar process to requesting images. The request URL in this case is: http://www.gravatar.com/digest.

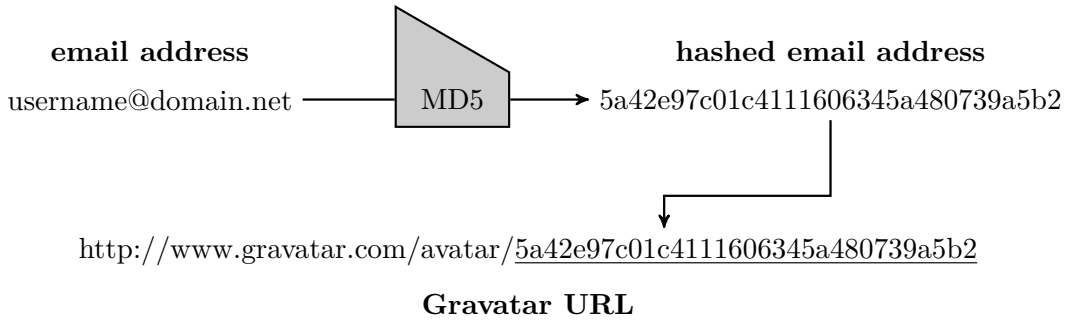

Figure 3.3: Gravatar URL generation from an email address.

[2] https://en.gravatar.com/
[3] http://www.redmine.org/
[4] https://www.drupal.org/

3.6.2 Re-identification

Gravatar often allows users to register and interact with a web service using a pseudonymous account based on their email address. However, the Gravatar request URL containing the MD5 digest is included in the web page and hence is public. A privacy attack on Gravatars consists in re-identifying the anonymous user's email address from the Gravatar digest, *i.e.*, the MD5 digest of the email address. Since, in most of the cases, the email address contains information about the user's name or initials, a user can be de-anonymized by cracking the MD5 digest.

It is worth noticing that some developers consider hashing (in particular MD5) to be sufficient for obfuscating email addresses. They further argue that emails are harder to crack than passwords since they are longer and less globally different from one another (see [Bon13]). On the contrary, email addresses could be easier to crack than properly generated passwords. We present the details below.

The first step of the re-identification attack consists in determining if we have a one-to-one or many-to-one mapping between the email addresses and the pseudonyms.

As specified in RFC 6531 [YM12], email addresses follow a specific structure: a local part and a domain part separated by the delimiter '@'. The domain part follows the naming rules of Internet domain names while the local part follows the rules specified in RFC 6531. More particularly, the local part has a maximum length of 64 characters which are limited to a subset of the ASCII characters: uppercase and lowercase Latin letters (a-z, A-Z); digits (0-9); special characters (`# - _ ~ ! $ & ' ( ) * + , ; = :`) as well as the period '.' as long as it is not the first or the last character. Other special characters are allowed but only used in specific cases that can be ignored in this study. As for the domain part, RFC 1035 [Moc87] allows for a string of up to 253 ASCII characters.

We first compute the total number of email addresses m. The theoretical value of m can be computed as $m = m_{local} \times m_{domain}$, where m_{local} and m_{domain} are respectively the number of possible local and domain parts. Taking a conservative approach, we consider only local parts composed of lower case Latin letters, digits, and '.', '-', '_' which lead to a total of $m_{local} = (38)^2(39)^{62} \simeq 2^{338}$ distinct values. As for the domain part, we again take a conservative approach by restricting the set of characters to lower case Latin letters, digits, and '-', '_'. The corresponding number of domains is $m_{domain} = 38^{253} \simeq 2^{1328}$. Hence, under these restrictions, the total number of email addresses is therefore $m = 2^{338} \times 2^{1328} = 2^{1666}$. Also, since the length of an MD5 digest is 128 bits, we have $n = 2^{128}$. As the theoretical value of m satisfies $m \gg n$, the mapping appears to be many-to-one. This implies that the average anonymity set of a Gravatar is $\frac{m}{n} = \frac{2^{1666}}{2^{128}} = 2^{1538}$, which is more than enough to provide strong anonymity guarantees. However, all along this computation, we implicitly assume that all the $m = 2^{1666}$ email addresses are in use and that they are evenly distributed in the afore-mentioned valid space.

The above assumptions are in fact not valid. First, the domain part is in effect restricted to a much smaller space. Indeed, as of today, there are roughly 326 million domain names registered in the world as reported by VERISIGN in its 2016 report [Ver15]. Moreover, according to another recent report [The15], the total number of worldwide email

accounts is roughly 4.7 billion, which means that the value of m in practice is roughly 2^{32} instead of the previously computed value of 2^{1666}. As a consequence, $m \ll 2^{64}$ and we have a one-to-one mapping between the emails addresses and their pseudonyms. Second, the domain part of popular email addresses generally fall into a much smaller space, corresponding to big email service providers such as Google (gmail.com), Yahoo! (yahoo.com) or ISPs such as AOL (aol.com), Verizon (verizon.net), Orange (orange.fr). Moreover, some domains are tightly linked to specific regions of the world (*e.g.*, yahoo.co.uk is mainly used by citizens of the UK). Third, the local part of the email address is generally chosen by its owner to be meaningful to others or to clearly identify an individual. As a consequence, they often contain real names, pseudonyms and dictionary words. A common pattern is the concatenation of the firstname and the lastname possibly separated by one of the following delimiters: '.', '-', '_'. For these reasons, despite the theoretically large size of the valid email address space, *"real"* email addresses fall into a much smaller space and are in general predictable and vulnerable to a variety of dictionary-based attacks.

A re-identification attack in practice against Gravatar was demonstrated by Bongard [Bon13]. The goal of the attack was to identify the email addresses of anonymous commentators on a French political blog (`fdesouche.com`). To this end, Bongard developed a custom crawler to acquire around 2,400 MD5 hashes of the commentators. The author then generated a dictionary of major email providers (including those providing disposable email addresses), cracking dictionaries (French and English names, Wikipedia entries, sports teams, places, numbers, birth years and postal codes), cracking rules (*e.g.*, most email addresses follow patterns such as firstname.lastname, firstname_lastname, *etc.*) and a cracking rig. The dictionary coupled with the password-cracking tool `oclHashcat` [Ste16] allowed to recover 45% of the email addresses. The author using another custom email cracking software based on a Bloom filter was able to recover 70% of the email addresses (an increase by 25%). In the context of Gravatar digest cracking, a Bloom filter was used to represent the dictionary, thereby requiring less memory than maintaining the dictionary in its native form. Moreover, its computational efficiency allows to recover more email addresses per unit time. The disadvantage of employing Bloom filters is that the cracking software now generates false positives, *i.e.*, it may claim to have cracked an MD5 digest while in effect it has not.

The idea of recovering email addresses from Gravatar digests was previously demonstrated in 2008 by a user called Abell[5] on `developer.it` — it was possible to recover 10% of the addresses of 80,000 `stackoverflow.com` users.

3.7 MAC Address Anonymization

A MAC address is a 48-bit identifier uniquely allocated to a network interface and by extension to a networked device. Because it uniquely identifies a device and its user it is a personally identifiable information and must therefore be considered as sensitive. MAC addresses are collected for tracking purposes by mobile applications [ACRF14] and Wi-Fi

[5] `http://bit.ly/2aMN007`

tracking systems [ME12]. Wi-Fi tracking systems monitor human activities by passively collecting information emitted by portable devices in an area of interest. Commercial applications of Wi-Fi tracking include retail-analytics and targeted advertisements in the physical world. These systems rely on the MAC address of Wi-Fi devices in order to detect and distinguish individuals. Indeed, most portable devices having their Wi-Fi interface enabled periodically broadcast packets containing their MAC address in order to discover surrounding Wi-Fi access points. As a consequence, Wi-Fi tracking systems store mobility traces of individuals along with a unique identifier, the MAC address, that could be potentially linked back to the owner of the device.

In an attempt to protect privacy of individuals monitored by those systems, anonymization techniques have been adopted. One of the most popular solutions is the application of a cryptographic hash function to the MAC address, *i.e.*, instead of storing a MAC address, the system stores the digest of the MAC address. This solution has the advantage of preserving a unique identifier for each individual (given the extremely low collision probability of those hash functions), while supposedly being impossible to be *"reverse-engineered [...] to reveal a device's MAC address."* [Euc15]. Hash-based anonymization of MAC addresses have therefore been widely adopted by major actors of the Wi-Fi tracking industry.

3.7.1 One-to-One Anonymization

The first attempt to anynomize MAC addresses was made using the SHA-1 hash function [Euc15]. Since, a MAC address is 48 bits long, we have $m = 2^{48}$ identifiers and $\ell = 160$ (length of a SHA-1 digest). For these parameters, each pseudonym is associated to a unique identifier. Moreover, since m is small enough, an attacker can easily mount the re-identification attack.

We develop the re-identification attack using `oclHashcat` [Ste16] and two GPU setups: one with an integrated GPU and the other with dedicated ATI R9 280X GPU. Table 3.1 displays the result of our experiments along with results from the benchmark of `oclHashcat`. Using the ATI R9 280X GPU, 2.6 days were enough to compute the hash of all possible MAC addresses. This shows that an exhaustive search is practical with off-the-shelf hardware and freely available software. Moreover, if the attacker has access to an AMD HD 6990 hardware, the cracking time can be reduced to a day.

It is even possible to speedup the attack by taking into account the structure of a MAC address (Figure 3.4). In order to guarantee the global uniqueness, MAC addresses are allocated to vendors in a range of 2^{24}, which is identified by a OUI (Organizationally Unique Identifier) prefix corresponding to the first three bytes of a MAC address. The remaining three bytes correspond to the network interface controller (NIC) which identifies an interface within a given OUI range.

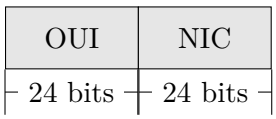

Figure 3.4: Structure of a MAC address.

Table 3.1: Computation time for 2^{48} SHA-1 digests using our own hardware (the first two rows) and the benchmark results from `oclHashcat` (the remaining rows). The number of hashes per second provides the average perfomance of the cracking tool on the concerned hardware. Cells marked * are estimated results. We used the number of hashes per second to estimate the time required to hash 2^{48} MAC addresses.

Hardware	# 10^6 hashes/s	Time (days)
Integrated GPU	11	296*
ATI R9 280X	1228	2.6
NVIDIA Quadro 600	80	41*
NVIDIA GTX 560 Ti	433	7.5*
NVIDIA GTX 570	629	5*
AMD HD 7970	2136	1.5*
AMD HD 6990	3081	1*

Currently, 22,317 OUI prefixes have been allocated by IEEE (the list is publicly available [IEE16]). It means that only 0.1% of all the possible MAC addresses can be encountered in practice. This observation reduces the exhaustive search from 2^{48} to $22317 \times 2^{24} \approx 2^{38}$ hash computations.

The discrimination attack for a single MAC address is instantaneous. It simply consists in computing the digest of the MAC address and checking for belonging to a specified set. Moreover, one can also exploit the structure of MAC addresses to discriminate devices. To this end, let us consider a scenario in which we have a database of MAC addresses corresponding to Wi-Fi devices in proximity to a certain location. The database is assumed to have been anonymized using a hash function. We further assume that the adversary knows the location from where the pseudonyms have been collected (such as a train station).

In order to de-anonymize the pseudonyms, the adversary exploits the distribution of OUIs among different vendors. Figure 3.5 shows the vendors having the highest number of OUIs registered to IEEE. Relying on the results of Figure 3.5, she can safely assume that most of these pseudonyms correspond to that of smartphones and Wi-Fi routers. Therefore, instead of mounting a brute force search with all existing OUIs, she first tries the MAC addresses with a OUI associated to popular vendors: Cisco, Apple and Samsung. Moreover, if the adversary wishes to know if a pseudonym corresponds to a Cisco device, then she needs to test 618×2^{24} MAC addresses. In case, she wishes to know if the pseudonym corresponds to a popular smartphone vendor, in particular Apple or Samsung, she would need to test $(474 + 317) \times 2^{24} = 791 \times 2^{24}$ MAC addresses.

In Table 3.2, we present the time required to learn whether a pseudonym corresponds to a given vendor. The results show that the top six vendors can be discriminated within 8 seconds. Clearly, for less popular vendors, the number of acquired OUIs should be much less than that of the top six and hence the required time to discriminate them should be even lower.

The attack reasoning is not limited to smartphones and network equipments. It can be extended to any device from the Internet-of-Things starting with drones (Parrot[6] has

[6] http://www.parrot.com

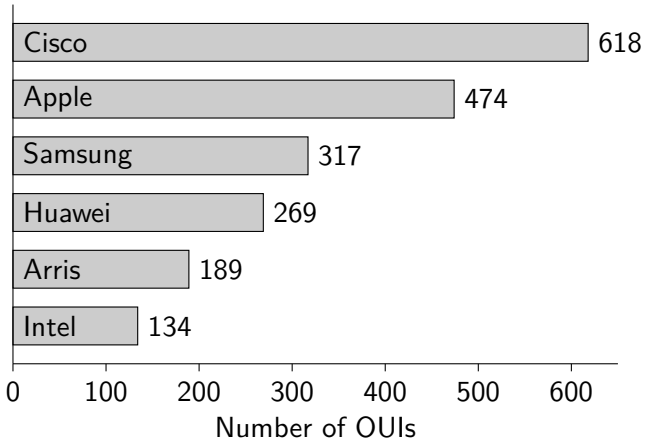

Figure 3.5: Number of OUIs per vendor (top 6).

Table 3.2: Time to discriminate the top 6 vendors in the OUI list using ATI R9 280X GPU.

Vendor	**Time** (s)
Cisco	8
Apple	7
Samsung	5
Huawei	4
Arris	3
Intel	2

five OUIs and DJI[7] has one) and Wi-Fi based fire alarm system (1 OUI for Firepower system[8]).

3.7.2 Many-to-One Anonymization

We now consider the case of anonymizing MAC addresses with truncated digests such that $m \gg 2^{\ell}$. We work with $m = 2^{38}$ (a consequence of the total number of allocated OUIs) and we assume that $\ell = 20$ (SHA-1 digests truncated to 20 bits). The parameters lead to a setting in which $m > n \log(n)$, hence a good approximation for the lower bound of the anonymity set size is $m/2^{\ell} = 2^{18}$. Intuitively, having a large lower bound of 2^{18} should guarantee an acceptable anonymity. However, we explain in the following why this intuition is wrong.

Let us consider an example anonymity set associated to the pseudonym `0xffffff`. It contains the addresses `000668CF92DF` and `000E5B53A051` among others. This is obtained by employing a brute force search on 2^{38} MAC addresses. The first address corresponds to a product of Vicon Industries Inc[9]. The vendor sells IP cameras for surveillance. The second address is a product of ParkerVision[10] which provides radio frequency solutions and wireless devices. For the anonymity set to be viable, one must have the pseudonym

[7] http://www.dji.com/
[8] www.firepower.ca
[9] http://www.vicon-security.com/
[10] http://parkervision.com/

`0xffffff` to be associated to `000668CF92DF` and `000E5B53A051` with the same probability. However, it is clear that in a database corresponding to wireless devices the address `000668CF92DF` has a probability 0 to occur. The opposite is true if we consider a database of wired devices. This side information can be explicitly given with the database or can be acquired by an adversary.

To extend the example, we need to add additional semantics to MAC addresses. We classify the OUI prefixes depending on the type of products the vendor manufactures. We define four labels: wireless, wired, both and unknown depending on whether the vendor manufactures wireless and/or wired devices. A label can be given to a vendor upon exploring the web page of the vendor when available. Otherwise, it can be based on the first result returned by a search engine. We employ Microsoft Bing[11] in our study as the search result can be obtained in the RSS format which makes the result easy to parse. To the best of our understanding, GOOGLE search engine does not provide this feature and hence the search response becomes difficult to parse.

Currently, 22317 OUI addresses are associated to 17365 vendors. Moreover, we have 92 OUI addresses associated with an unknown vendor name (private in the database). We crawled the websites of these vendors whenever possible and built a database of their label. We do so by crawling a maximum of 6 pages (chosen arbitrarily to not overload the server) from the fully qualified domain name corresponding to the vendor or to that of the first result returned by Bing. Our database has labels for 12201 vendors in total (70% of 17365). The breakdown for the labels is shown in Figure 3.6.

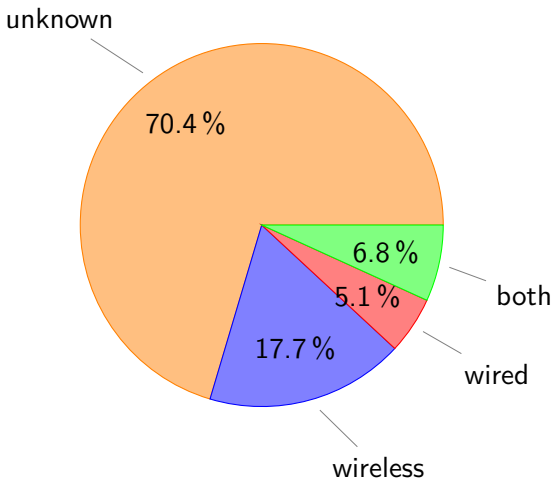

Figure 3.6: Percentage of vendors of each type.

In total, the anonymity set of the pseudonym `0xffffff` contains 14103 MAC addresses which are associated to wired devices. Theses addresses are unlikely to be associated with the pseudonym `0xffffff` in a wireless related database. Hence, the anonymity set size is not 2^{18} but $2^{18} - 14103 = 24801 = 2^{17}$, *i.e.*, a size reduction by a factor of 2. It is certainly possible to reduce it further by improving the classification and by adding more semantics to OUI prefixes, for instance the country in which the vendor is based.

[11] http://www.bing.com/

These results show that that controlling the anonymity set size is very difficult. We certainly can not assume $m = 2^{38}$, rather it must be adapted to the dataset we need to anonymize. Additionally, data custodians must also choose the correct ℓ that yields an acceptable anonymity set size for a given dataset.

3.8 Summary

In this chapter, we study the anonymity achieved when an identifier is hashed using a cryptographic hash function to generate a pseudonym. We use the balls-into-bins framework to estimate the anonymity set size. We explore its application to two usecases namely, MAC address anonymization and Gravatar email address anonymization. Another usecase of Safe Browsing has been left to Chapter 7. These applications demonstrate the limits of cryptographic hash functions to create pseudonyms. A more advanced application of hashing for anonymization could be to use a keyed-hash function *aka* a hash-based message authentication code (HMAC). Employing an HMAC may counter the re-identification and discrimination attacks since the underlying key cannot be guessed by the attacker. However, use of a key drastically restricts its deployment as it should be securely stored and might need to be exchanged with a possibly malicious entity. Moreover, since an HMAC is deterministic, two records for the same identifier can be linked. Other attacks exploiting auxiliary information have also been identified in the past [opi14]. In the next chapter, we study how Bloom filters can give better privacy for data collection and computation. Furthermore, the considered frameworks provide provable guarantees on the level of privacy achieved.

Chapter 4

Privacy Provisions of Bloom Filters

Contents

4.1 How not to use a Bloom Filter **48**
4.1.1 Naive Approach 48
4.1.2 Auditing Without Full Hashes: Blackhash 48
4.2 Bloom Filters for Better Privacy **49**
4.2.1 Differential Privacy 50
4.2.2 BLIP 50
4.2.3 RAPPOR 52
4.3 Bloom Filters for Private Membership Queries **53**
4.3.1 Context and Problem Statement 53
4.3.2 A Survey of Alerting Websites 54
4.4 Privacy-Friendly Alerting Websites **57**
4.5 Solutions for Private Databases **58**
4.6 Solutions for Public Databases **59**
4.6.1 Private Information Retrieval 59
4.6.2 Private Membership Query Using PIR 60
4.6.3 Extension with PBR Protocol 62
4.7 Practicality of the Solutions **63**
4.7.1 Applicability of PIR 63
4.7.2 Experimental Analysis 64
4.8 Related Work **66**
4.9 Summary **67**

We have seen in the previous chapter, how employing plain hashing to ensure user's privacy can fail drastically. In this chapter, we extend our study to explore the privacy provisions of Bloom filters. To this end, we explore three different use cases. The first use case shows how Bloom filters should not be used. The next one gives an overview of

two state-of-the-art privacy-preserving tools namely, BLIP [AGK12, AGMT15] and RAPPOR [EPK14]. These tools are built atop Bloom filters. For the third use case, we present a novel application of Bloom filters to privately query a leaked database.

4.1 How not to use a Bloom Filter

Let us consider the problem of *password auditing* — a preventive mechanism to resist password cracking tools such as John the Ripper[1] and Hashcat[2]. In its restricted form, password auditing consists in determining whether any of the system passwords are weak and hence susceptible to cracking tools. This is essentially performed with the help of an auditor who uses dictionary based tools to filter weak digests.

In the following, we present existing approaches to password auditing of this kind and analyze their weaknesses.

4.1.1 Naive Approach

A naive approach to password audit would typically involve a system administrator who extracts password digests from systems and then sends them to a third-party security auditor or an in-house security team. The auditor relying on tools such as John the Ripper or Hashcat may easily uncover potentially weak passwords. However, such an approach ensues serious risks. The password digests may be lost or stolen from the audit team. Furthermore, a rogue audit team member may secretly make copies of the password digests and may mount *pass-the-hash attacks*. In a pass-the-hash attack, an attacker possessing only a password digest can pass the authentication step in a system without requiring to have the password in clear. Several systems such as Windows Authentication Protocols are vulnerable to these attacks.[3] Worse, some of these digests may correspond to easy-to-crack passwords. The auditor may recover in clear the weak passwords and use it for malicious purposes before reporting it back to the system administrator.

Consequently, it is hard to guarantee that the password digests are handled and disposed of securely and that access to the digests is not abused. Indeed, only the system administrator and his team should have access to password digests. Extracting the digests and giving them to someone else fundamentally compromises the security of the system.

4.1.2 Auditing Without Full Hashes: Blackhash

Blackhash [Til14] was a tool (discontinued since 2015) designed for restricted auditing of passwords, *i.e.*, check for weak password digests in the system file without having access to the full digests. It works by building a Bloom filter that contains the system password digests. The system administrator extracts the password digests and then uses Blackhash to build the filter. The filter is saved to a file, then compressed and given to the audit team. The audit team maintains a set of dictionaries of weak passwords against which the

[1] http://openwall.com/john/
[2] http://hashcat.net/oclhashcat/
[3] http://passing-the-hash.blogspot.ca/

password digests are to be tested. Upon reception of the filter, the auditor simply checks for each entry of the dictionary, whether or not it is present in the filter. If weak passwords are found to be present in the filter, the audit team creates a weak filter of these passwords and sends it back to the system administrator. Finally, the system administrator tests the weak filter against the system digests to identify individual users with weak passwords.

Privacy issues with Blackhash. Moving from a list of password digests to a sequence of bits in a Bloom filter ostensibly obfuscates the passwords. However, developers claimed that Blackhash does not reveal password digests to the auditor. Hence, it constitutes a better and secure tool compared to the naive approach.

However, this reasoning gives a false sense of security. Indeed, a Bloom filter is not simply a sequence of random bits but represents a well-defined data structure and may reveal a lot more of information than intended, when it falls into the hands of an adversary. In fact, contrary to the claim, using a Bloom filter of password digests instead of full digests does not improve user's privacy. The most serious issue with Blackhash is that the auditor while finding the weak passwords with the help of dictionaries actually retrieves the weak passwords in clear. To paraphrase, Blackhash requires the system administrator to reveal the weak passwords and hence gives similar privacy as achieved by the naive approach.

The only difference with respect to the naive approach is that in addition to the actual weak passwords in the list of digests, the auditor also obtains some passwords that are false positives. We note that there is no mechanism for the auditor to determine whether a recovered password is a false positive or a true positive (an entry in the initial list of digests). This hence introduces some noisy results in the list of recovered passwords that is otherwise absent in the naive approach.

The lesson learned here is the following: A naive use of Bloom filters to hide items is error-prone as Bloom filters is not a method of obfuscation *per se*. In the following, we discuss two state-of-the-art tools that use Bloom filters along with randomization to achieve strong privacy guarantees.

4.2 Bloom Filters for Better Privacy

In this section, we present BLIP [AGK12, AGMT15] and RAPPOR [EPK14]. In BLIP, the main objective is to privately compute in a distributed manner the similarity between user profiles by relying only on their Bloom filter representations. While, RAPPOR is a tool developed by Google to gather usage statistics. Both these tools use randomization to turn Bloom filters into a tool useful for private computations.

In the following, we first give an overview of Differential Privacy [Dwo06], a strong privacy guarantee that BLIP and RAPPOR aim to achieve. And, in the sequel we present the two tools.

4.2.1 Differential Privacy

Differential privacy is a privacy notion put forth by Dwork [Dwo06] in the context of privacy-preserving data mining on statistical databases.

Differential privacy was conceived for scenarios where a data holder or a data curator asks for data from different subjects and wishes to release the result of the collected database to the public — which in turn may query it for useful statistical information, while preserving the privacy of each subject. It describes the following promise, made by the data holder/curator to a data subject: The subject will not be affected, adversely or otherwise, by allowing his/her data to be used in any study or analysis, no matter what other studies, data sets, or information sources, are available. The promise is held by ensuring that even if the subject changes his/her data, then the output of the analysis should not change much. This for instance implies that the data subject can pretend to have provided any other permissible data without affecting too much the analysis. Seen this way, differential privacy can be viewed as a model of *"plausible deniability"* to consider scenarios where the data collection was either done with bad intent or to cope with future incidents when the curator may obtain additional auxiliary information on the subject and attempt to breach the subject's privacy.

Differential privacy aims at providing strong privacy guarantees with respect to the input of some computation by randomizing its output. The guarantee is independent of the auxiliary information that the adversary may have gathered. In more formal terms, differential privacy is a property of a randomized procedure $\mathcal{A}$ which one may run on a dataset $X = (x_1, \ldots, x_n)$ producing some output $\mathcal{A}(X)$. $\mathcal{A}$ is then ϵ-*differentially* private if for any two neighbour datasets X and X' which differ in just one record, *i.e.*, $X = X'$ except that $x_i \neq x'_i$ for some $1 \leq i \leq n$, the following inequalities hold:

$$\text{for all outcomes } v, \quad e^{-\epsilon} \leq \frac{\mathbb{P}(\mathcal{A}(X) = v)}{\mathbb{P}(\mathcal{A}(X') = v)} \leq e^{\epsilon}$$

The probability is computed over the randomness in $\mathcal{A}$.

The parameter ϵ is public and may take different values depending on the application. The smaller the value of ϵ, the higher the privacy but also, as a result, the higher the impact might be on the utility of the resulting output.

Several generic randomization techniques have been studied that achieve ϵ-differential privacy for a function f by considering $\mathcal{A}$ that adds random noise to the true answer of f before releasing it. This includes the *Laplacian mechanism* [DMNS06] and the *Exponential mechanism* [MT07]. Interested readers may consult the references for details on these mechanisms.

4.2.2 BLIP

BLIP [AGK12, AGMT15] considers the model of a distributed system of nodes, in which nodes are characterized by their profiles representing the associated node's interests. For example, the profile of a node can be a vector of queries that have been performed on a search engine or ratings on movies.

The general goal is to compute a similarity measure of profiles between pairs of nodes. However, as each profile is personal and may reveal private attributes about the node, the profiles must remain private during the computation. The computation in BLIP is done using a randomized version of a Bloom filter. In fact, each node stores its profile in a filter which is later used for private computations. As we have seen in the case of Blackhash, it is also desirable that this computation should restrict the possibility for an adversary observing the Bloom filter to infer the presence or absence of a particular item in this profile.

The approach developed in BLIP relies on bit flipping to achieve differential privacy. BLIP in fact stands for *Bloom-then-flip*. First, each node builds a Bloom filter that represents its profile. Second, each bit of the filter is flipped with a chosen probability $p < \frac{1}{2}$, *i.e.*, a bit b of the filter is set to $\bar{b} = 1 - b$ with probability p. Third, the generated filter is then exchanged with other nodes. These filters are then used to compute a similarity measure. We note that the inner product of two filters measures the similarity between the sets that they represent. Hence, by computing the inner-product, the nodes can learn the similarity between their profiles.

Figure 4.1 schematically presents these procedures for a sample Bloom filter.

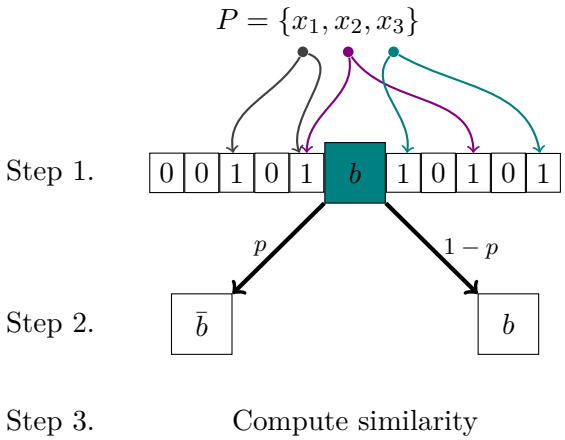

Figure 4.1: BLIP steps for a profile P with three items. The filter is built using 2 hash functions and a flip probability of p. Flipping of a bit b in the filter is shown in Step 2.

The randomization procedure of BLIP has been proved to be ϵ-differentially private if the flipping probability verifies:

$$p = \frac{1}{1 + e^{\frac{\epsilon}{k}}}.$$

The example of Figure 4.1 with $p = \frac{1}{3}$ and $k = 2$ is 1.4-differentially private.

The utility of BLIP is measured as how close the similarity between two flipped filters is to the actual similarity between the filters where none of the bits are flipped. Alaggan *et al.* in [AGK12] experimentally evaluate the trade-off between privacy and utility that can be reached with BLIP on three datasets. For proofs and more discussion on the utility obtained by BLIP, interested readers may refer to [AGK12].

4.2.3 RAPPOR

RAPPOR [EPK14] is another privacy-preserving tool that achieves differential privacy. The underlying idea is very similar to BLIP. It was designed to crowdsource statistics from an end-user client software. For instance, it can be used to obtain statistics on certain browser settings. In fact, it was tested with the Chrome browser to obtain usage reports.

We present a simplified version of RAPPOR (*aka* one-time RAPPOR) which still guarantees differential privacy. To this end, let us consider a scenario where users wish to send a data represented in the form of a string to a distant server who wishes to compute statistics on the string, *e.g.,* computing the frequency of each string. Similar to BLIP, each user first builds a Bloom filter with parameters that depend on the level of privacy one wishes to guarantee. The data is then inserted into the filter. The randomization mechanism in RAPPOR is only slightly different compared to BLIP. The randomization mechanism has a probability parameter p. Each bit of the filter is then set to 1 with a probability $p/2$, it is set to 0 with a probability $p/2$ and it is left as it is with a probability $1-p$. The resulting filter is then sent to the server for statistics. This randomized procedure has been shown to be ϵ-differentially private when the filter parameter k satisfies:

$$\epsilon = 2k \ln \left(\frac{1 - \frac{1}{2}p}{\frac{1}{2}p} \right).$$

A schematic representation of this randomization scheme is presented in Figure 4.2.

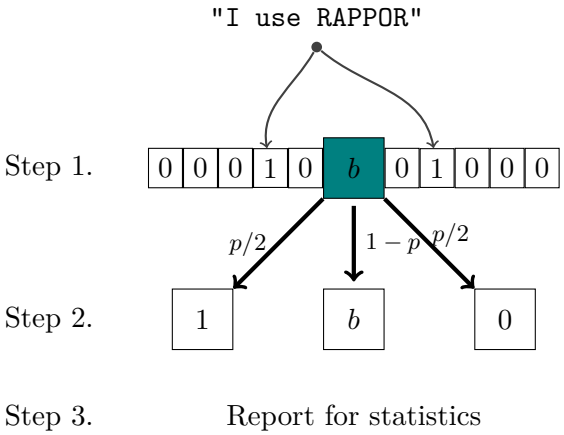

Figure 4.2: Steps in RAPPOR for the string `"I use RAPPOR"`. The filter is built using 2 hash functions and with a flip probability p. Flipping of a bit b in the filter is shown in Step 2.

Once the filter is randomized, it can be sent to the server to compute utility such as aggregate statistics. We note that the algorithm to compute the frequency of a string from the aggregate set of randomized filters requires advanced machine learning techniques. Due to the involved nature of these algorithms, we do not discuss them here, but rather refer the interested readers to the original paper by Erlingsson *et al.* [EPK14].

4.3 Bloom Filters for Private Membership Queries

In the following, we present a relatively new problem where user's privacy is desired, and again hashing can not be a solution. While hashing fails, Bloom filters apparently come to our rescue. We employ Bloom filters as a tool together with existing privacy-preserving cryptographic protocols to provide a solution with strong privacy guarantees.

4.3.1 Context and Problem Statement

In the recent years, we have witnessed an increasing number of data leaks from major Internet sites including Adobe, Snapchat, LinkedIn, eBay, Apple and Yahoo!. (see bit.ly/19xscQO for more instances). While in most of the cases passwords' files have been targeted, database of identifiers, phone numbers and credit card details have also been successfully exfiltrated and published. These leakages dealt a substantial blow to the trust of people in computer security. Even worse, the data leaks have become a ready source of income for several security companies. To cite an example, in August 2014, over a billion records were discovered by Hold Security, which include confidential material gathered from 420,000 websites, including household names, and small Internet sites. Hold Security charges users/companies wishing to check if they were a victim of this breach.

The aftermath of these leakages has led to three pivotal developments. First, the bad security policies of major websites have been exposed, and better policies have been proposed to survive leakages (see [PMW+09, KAPK13, JR13]). In [PMW+09], Parno *et al.* design an architecture to prevent database leakage. At CCS 2013, Kontaxis *et al.* propose SAuth [KAPK13], an authentication scheme which can survive password leakage. At the same conference, Juels and Rivest present the Honeywords [JR13] to detect if a passwords' file has been compromised.

Second, security community has obtained datasets to study the password habits of users. In [DBC+14], Das *et al.* consider several leaked databases to analyze password reuse. De Carnavalet *et al.* [dM14] use these datasets to test the effectiveness of password meters.

Third, a new kind of website has appeared: *alerting website*. Users can check through these sites whether their accounts have been compromised or not. In order to check whether a user is a victim of data leakage, alerting websites ask for an identifying data such as username or email address and sometimes even a password. These websites are maintained by security experts such as haveibeenpwned.com by Troy Hunt, security companies *e.g.*, LastPass, and even government institutions such as the German Federal Office for Information Security (bsi): sicherheitstest.bsi.de. The case of the Snapchat leakage is particularly instructive to understand alerting websites. In December 2013, private information concerning 4.6 million accounts were leaked. The database first appeared on SnapchatDB.info (owned by the hackers) in January 2014. The website allowed any user to download the database dump in its entirety. This site was soon suspended without any official justification and several sites have taken over SnapchatDB.info. As they allowed

complete database download, it is apparent that they lacked users' privacy considerations.

On the one hand, these websites are very useful in alerting users, while on the other hand, they are real "*booby traps*". The problem is the following: when a user submits a username or an email address or a password, the site searches whether it exists or not in the leaked database. If it exists, the user is warned and the website has accomplished its purpose. However, if it is not present in the database, the site owner learns for free a username/email address/password.

Most of these sites advert to the users that they do not indulge in phishing activities but this is the only guarantee available to the user. The goal of alerting websites is to reduce the effect of data leakage but not to amplify it! Considering the risks of using alerting websites, we naturally raise the following question: How to design alerting websites which cannot be turned into a phishing trap? The user must have a guarantee that it is not possible for the database owner to collect his information during a search query. As seen in the previous chapter, sending a hash of the personal data serves no good as personal data can be easily guessed due to their inherent low entropy.

Contribution. We make the following contributions in the study and analysis of alerting websites.

1. We first examine 17 popular alerting websites (Section 4.3.2) and analyze their working mechanism, and their approach to deal with privacy. Our findings reveal that several of these websites have huge phishing potential and hence users should be careful while visiting any of these websites.

2. We study solutions for designing *privacy-friendly alerting websites*. Two different scenarios have been considered depending on whether or not the database is public. In case of private database (Section 4.5), *private set intersection protocol* [DCGT12], and its variant *private set intersection cardinality protocol* [DCT10], yield an immediate solution. The scenario of public database (Section 4.6) requires us to adapt *private information retrieval protocol* [CGKS95], for membership testing. This is achieved by combining it with Bloom filters. These protocols subsumed under the name of *Private Membership Query* protocols ensure user's privacy in the *honest-but-curious* model.

3. We experimentally analyze the merits of these solutions with respect to real world data leakages (Section 4.7). Our evaluation criteria include memory and time efficiency, communication cost in terms of bandwidth and scalability.

4.3.2 A Survey of Alerting Websites

Users can be alerted on the fact that their account or personal information has been leaked. Figure 4.3 presents a sample alerting website for LinkedIn 2012 data leak of 6.5 million passwords. Note that the service asks for users' passwords.

In this section, we discuss the characteristics of these websites which evidently offer opportunities for sophisticated phishing attacks. Websites alerting users about their

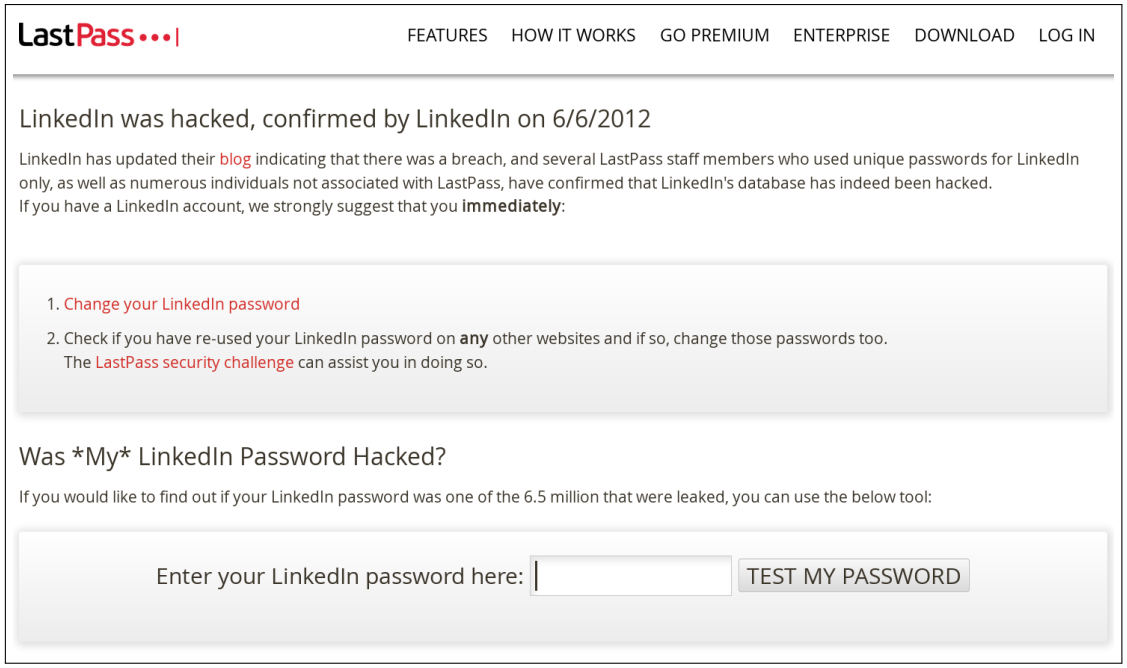

Figure 4.3: A sample alerting website: https://lastpass.com.

account or data leakage can be divided into three types according to their sources of information.

Single-source (S). Some websites are associated with a single data leakage. This is the case for adobe.cynic.al, bit.ly/1by3hd9, lucb1e.com/credgrep and adobe.breach.il.ly. These websites are related to the ADOBE data leakage of 153 million accounts which occurred in October 2013. The last three websites were successfully tested on 22/12/2014 but can not be accessed anymore. Other websites for instance snapcheck.org,[4] findmysnap.com, and lookup.gibsonsec.org are similarly associated with the SNAPCHAT leakage (4.6 million usernames, phone numbers and cities of residence exposed in January 2014).

Aggregator (A). We observe that 5 of these websites search through several databases to inform users if their data has been exposed. For instance, shouldichangemypassword.com (bit.ly/1aJubEh for short) adverts to use 3346 leaked databases while only 194 are officially known. The remaining four are maintained by private companies: lastpass.com, bit.ly/1fj0SqV, bit.ly/1aJubEh and dazzlepod.com/disclosure. The last remaining site haveibeenpwned.com is designed and maintained by a security expert named Troy Hunt.

Harvester (H). Three sites claim to have created their own databases from harvested data. Two of these are maintained by famous security companies hacknotifier.com and pwnedlist.com/query. The last site is maintained by the German Federal Office for Information Security (BSI).

[4]The database cannot be accessed anymore (10/12/2014).

The rue89.nouvelobs.com site is slightly different from the others. In September 2014, this French news website bought from Darknet, 20 million French email addresses for 0.0419 bitcoins (see article bit.ly/1lKAxsB). The article offers the opportunity to check if the reader's addresses are included in the leak.

Table 4.1: Analysis of 17 alerting websites (* result as on 22/12/2014).

Websites	Type	Database(s)	https	Statement	Answer	Descrip.
rue89.nouvelobs.com	S	Unknown	✗	✓	✓	✗
adobe.cynic.al	S	Adobe	✗	✗	✓	✓
lucb1e.com/credgrep*	S		✗	✗	✗	✗
adobe.breach.il.ly*	S		✗	✗	✗	✗
bit.ly/1by3hd9*	S		✓	✓	✓	✗
snapcheck.org	S	SnapChat	✗	✗	✓	✗
findmysnap.com	S		✗	✗	✓	✗
lookup.gibsonsec.org	S		✗	✗	✓	✗
didigetgawkered.com	S	Gawker	✗	✗	✗	✗
dazzlepod.com/disclosure	A	28	✗	✓	✓	✗
lastpass.com	A	6	✓	✓	✓	✗
haveibeenpwned.com	A	9	✓	✓	✓	✓
bit.ly/1fj0SqV	A	12	✓	✗	✗	✗
bit.ly/1aJubEh	A	3346/194	✓	✗	✓	✗
hacknotifier.com	H	Unknown	✗	✓	✓	✗
pwnedlist.com/query	H	Unknown	✓	✗/✓	✓	✓
sicherheitstest.bsi.de	H	Botnets	✓	✓	✓	✗

We have reviewed 17 alerting sites and our findings are summarized in Table 4.1. To decide if a user can trust the service offered by these sites, we have considered four criteria:

1. The usage of a secure connection through HTTPS.
2. The existence or not of a security/privacy statement.
3. The fact that the site responds or not with an answer.
4. A technical description of all the operations performed on the data.

From Table 4.1, we observe that ten of these sites do not use HTTPS which means that the traffic towards them can be easily eavesdropped. Single-source alerting sites are the least trustworthy of all since most of them do not publish a privacy statement. The website bit.ly/1by3hd9 is a notable exception. Aggregator sites in general perform better. Most of them use HTTPS and have a statement concerning privacy or phishing. The website haveibeenpwned.com even has a description of how it works.

The harvesters are more controversial: hacknotifier.com claims that "*we use a 256-bit secured and encrypted SSL connection*", but does not use HTTPS or any encryption.[5] The website pwnedlist.com/query claims that "*this is not a phishing site*", but they also state (pwnedlist.com/faq) that "*Over the past years we've built an advanced data harvesting infrastructure that crawls the web 24/7 and gathers any potentially sensitive data ...*".

[5] Actually, subscribing for the hacknotifier.com watchdog is also not secure.

Four sites do not give any answer: either they are not functional anymore (like lucb1e.com/credgrep) or they are real phishing traps.

Almost all the sites receive account information in clear. However, there are two notable exceptions lastpass.com and dazzlepod.com/disclosure. The former uses cryptographic hash functions and truncation to obfuscate the query and seems to be the most transparent and trustworthy of all. Table 4.2 presents a summary of our observations on lastpass.com. The latter source, dazzlepod.com/disclosure only recommends to truncate the email address. With pwnedlist.com/query, it is also possible to submit the SHA-512 digest of the email address instead of the address itself.

Cryptographic hash functions, *e.g.*, MD5, SHA-1 or SHA-3 are however not enough to ensure the privacy of passwords, identifiers or email addresses: these data do not have full entropy. As we have seen in Chapter 3, email addresses were recovered from Gravatar digests, passwords too have been recovered from digests (see [NS05] for instance). Apple's **U**nique **D**evice **ID**s *aka* UDIDs are no exceptions. They are computed by applying SHA-1 on a serial number, IMEI or ECID, the MAC address of Wi-Fi and the MAC address of Bluetooth. The values used to produce a UDID can be guessed and LastPass asks only for the first 5 characters of UDID. It reduces the amount of information submitted to the site but the user is not warned if he provides more than 5 characters.

Table 4.2: Detailed analysis of `lastpass.com`.

Victim	Query	Policy	Privacy method
Adobe	Email	non-storage	None
LinkedIn	password	non-storage and non-logging	SHA-1
Snapchat	user name	non-storage and non-logging	SHA-1
Apple	UDID		Truncation
Last.fm	password	non-storage and non-logging	MD5
eHarmony	password	non-storage and non-logging	MD5

As a general conclusion, the measures taken by these websites such as hashing or truncation are clearly not adequate to ensure the privacy of users' queries. In the remainder of this chapter, we evaluate how existing cryptographic and privacy-preserving primitives can solve the problems associated to alerting websites. These privacy-friendly solutions should guarantee that the websites cannot harvest any new data from a user's query.

4.4 Privacy-Friendly Alerting Websites

As previously discussed, the existing alerting websites in general do not respect the privacy of a user and entail huge phishing potential. The need of the hour is to design *privacy-friendly alerting websites*. These websites would rely on what we refer to as *Private Membership Query* protocols — allowing a user to privately test for membership in a given set/database. Such a protocol would guarantee that no new data can be harvested from a user's query.

To this end, two different privacy objectives can be defined depending on the privacy policy of the database owner. One that we henceforth refer to as *Private Membership*

Query to Public Database, and the other as *Private Membership Query to Private Database*. This classification arises due to the fact that most of these leaked databases are available on the Internet (as hackers have acquired the database dump) and hence can be considered as public in nature. However, even though they are public in terms of availability, an ethical hacker might want to ensure that the leaked information is not used for malicious purposes and hence the database cannot be accessed in a public manner to consult private information corresponding to other users. Rendering the database private could also be of interest for government agencies such as the BSI.

We highlight that a private membership query protocol provides a direct solution to the problem of designing privacy-friendly alerting websites. A user wishing to know whether his data has been leaked would be required to invoke the private membership protocol with the database owner and learns whether he is a victim of the breach. Thanks to the user's privacy provided by the protocol, no new data can then be harvested by the website. Consequently, in the rest of this work, we concentrate on evaluating solutions for private membership query problem. In the sequel, we formalize the privacy policies and examine viable solutions in the two database scenarios.

4.5 Solutions for Private Databases

The scenario of private membership query to private database involves a private database $\mathcal{DB}$ and a user $\mathcal{U}$. The database $\mathcal{DB} = \{y_1, \ldots, y_n\}$, where $y_i \in \{0,1\}^\ell$ consists of n bit-strings each of length ℓ. User $\mathcal{U}$ owns an arbitrary string $y \in \{0,1\}^\ell$. Private membership query to $\mathcal{DB}$ consists in knowing whether or not user's data y is present in the database while keeping y private to the user and $\mathcal{DB}$ private to the database.

Adversary model: The client and the database-owner are supposed to be *honest-but-curious*, *i.e.*, each follows the protocol but tries to learn information on the data held by the other player.

The above problem is very closely related to the problem of *Private Set Intersection*, hence we examine its applicability to designing privacy-friendly alerting websites.

Private Set Intersection (PSI). PSI protocol introduced by Freedman *et al.* [FNP04] considers the problem of computing the intersection of private datasets of two parties. The scenario consists of two sets $\mathcal{U} = \{u_1, \ldots, u_m\}$, where $u_i \in \{0,1\}^\ell$ and $\mathcal{DB} = \{v_1, \ldots, v_n\}$, where $v_i \in \{0,1\}^\ell$ held by a user and the database-owner respectively. The goal of the user is to privately retrieve the set $\mathcal{U} \cap \mathcal{DB}$. The privacy requirement of the scheme consists in keeping $\mathcal{U}$ and $\mathcal{DB}$ private to their respective owner. Clearly, the private membership query to private database problem reduces to PSI for $m = 1$.

There is an abounding literature on novel and computationally efficient PSI protocols [KS05, HL08, JL09, DSMRY09, DCT10, HN12, HEK12, DCW13]. The most efficient protocols are the ones by De Cristofaro *et al.* [DCT10], Huang *et al.* [HEK12] and Dong *et al.* [DCW13]. The general conclusion being that for security of 80 bits, protocol by De Cristofaro *et al.* performs better than the one by Huang *et al.*, while for higher security level, the latter protocol supersedes the former. The most efficient of all is the protocol

by *Dong et al.* as it primarily uses symmetric key operations. We note that the communication and the computational complexity of these protocols is linear in the size of the sets.

Private Set Intersection Cardinality (PSI-CA). PSI-CA is a variant of PSI where the goal of the client is to privately retrieve the cardinality of the intersection rather than the contents. While generic PSI protocols immediately provide a solution to PSI-CA, they however yield too much information. While several PSI-CA protocols have been proposed [FNP04, KS05, HW06, VC05], we concentrate on PSI-CA protocol of De Cristofaro *et al.* [DCGT12], as it is the most efficient of all. We also note that PSI-CA clearly provides a solution to the membership problem: if the size of the intersection is 0, then the user data is not present in the database.

A possible private membership query protocol in the case of private database can be built by combining PSI/PSI-CA and Bloom filters. The idea is to build a Bloom filter $\vec{z}$ corresponding to the database entries and generate the set $\mathcal{DB} = \text{supp}(\vec{z})$. The client on the other hand generates $\mathcal{U} = \{h_1(y), \ldots, h_k(y)\}$ for a data y. Finally, the client and the database owner invoke a PSI/PSI-CA protocol to retrieve the intersection set/cardinality of the intersection respectively. However, this solution is less efficient than a PSI/PSI-CA protocol on the initial database itself. The reason being the fact that, with optimal parameters the expected size of $\text{supp}(\vec{z}) = m/2 = 2.88kn$ (*Cf.* Chapter 2) . Hence, the number of entries of the database in PSI/PSI-CA when used with Bloom filter is greater than the one of the original database.

4.6 Solutions for Public Databases

This scenario is modeled using a public database $\mathcal{DB}$ and a user $\mathcal{U}$. The database as in the previous scenario is $\mathcal{DB} = \{y_1, \ldots, y_n\}$, where $y_i \in \{0,1\}^\ell$. User $\mathcal{U}$ owns an arbitrary string $y \in \{0,1\}^\ell$ not necessarily in $\mathcal{DB}$. Private membership query consists in knowing whether or not user's data y is present in the database while keeping y private to the user.

The difference to the previous problem (Section 4.5) is that the database in this context is public. This leads to a trivial solution ensuring absolute privacy consisting in sending the database to the user, who using the available resources performs a search on the database. With huge databases of order GB, the trivial solution is not the most desirable one for low memory devices. In this scenario, a user would wish to securely outsource the search to the database-owner. In the following, we present tools which provide a solution in the public database case.

4.6.1 Private Information Retrieval

In the first place we present a protocol called *Private Information Retrieval* [CGKS95], which is the closest to our needs. In the sequel we use this protocol together with Bloom filters to obtain a protocol for private membership query to public databases.

PIR first introduced in the seminal work by Chor *et al.* [CGKS95] is a mechanism

allowing a user to query a public database while keeping his intentions private. In the classical setting of PIR [CGKS95], a user wants to retrieve the bit at index $1 \leq j \leq n$ in a database $\mathcal{DB} = \{y_1, \ldots, y_n\}$, where $y_i \in \{0, 1\}$, but does not want the database owner to learn j.

Adversary model: The database owner is supposed to be honest-but-curious.

Since the work by Chor *et al.*, several variants of PIR have been studied which include Private Block Retrieval (PBR) scheme – where the goal is to retrieve a block of bits instead of a single bit and **P**rivat**E** **R**etrieval by **K**e**Y**words (PERKY) [CGN98] – where the user only holds a keyword kw instead of an index j. While PIR may either be built on single or replicated database copies, most of the latter works only consider the more realistic single database scenario. These works improve on the communication complexity [Cha04, CMS99], [GR05], [KO97, Lip05]. The current best bound of $\mathcal{O}(\log^2 n)$ is independently achieved in [Lip05], [GR05]. In this work, we only consider single database protocols. The principle reason being that in our context a user interacts with only one website.

Clearly, the PIR problem is almost the complement of our problem. In private query to public database, user has a data and wishes to know the index (if it exists) where it appears in the database, while in PIR the user has the index and wishes to know the data stored at this index. However, they are not exactly the complement since our problem also considers non-membership queries while PIR intrinsically does not deal with it.

We note that despite the similarity of the two problems: PIR and private membership to public database, PIR stand-alone does not provide a solution to our problem. Classical PIR *per se* cannot be applied to our context since the user holding a data (email address, password, *etc.*) present in a database does not know its physical address in the database. Furthermore, PIR does not support non-membership queries as the database is constructed in a predefined manner and has only finite entries, while the set of all possible queries is infinite.

However, we show in the following section, that when combined with a Bloom filter, PIR behaves as a private membership query protocol.

4.6.2 Private Membership Query Using PIR

The intuition behind the protocol is simple: PIR does not answer non-membership queries, but a Bloom filter does. Hence, we propose to combine PIR and Bloom filters to get the desired functionality. This can indeed be achieved by using PIR as a query protocol, whereby the database representation is changed to Bloom filters. In order to ease understanding of the actual protocol, we first present a membership query protocol to a Bloom filter. The protocol in itself comes with very little if no privacy, but can be easily modified to build another protocol with strong privacy guarantees.

Let us assume that *Alice* wants to check if her value y is included in the Bloom filter $\vec{z}$ held by *Bob*. The easiest way to do so consists for Alice to send y to Bob. Bob queries the filter on input y. He then sends 0 or 1 to Alice as the query output. If the canal between Alice and Bob has limited capacity, another strategy is possible and is described in Figure 4.4. Alice cannot send y due to some channel constraints but she can send

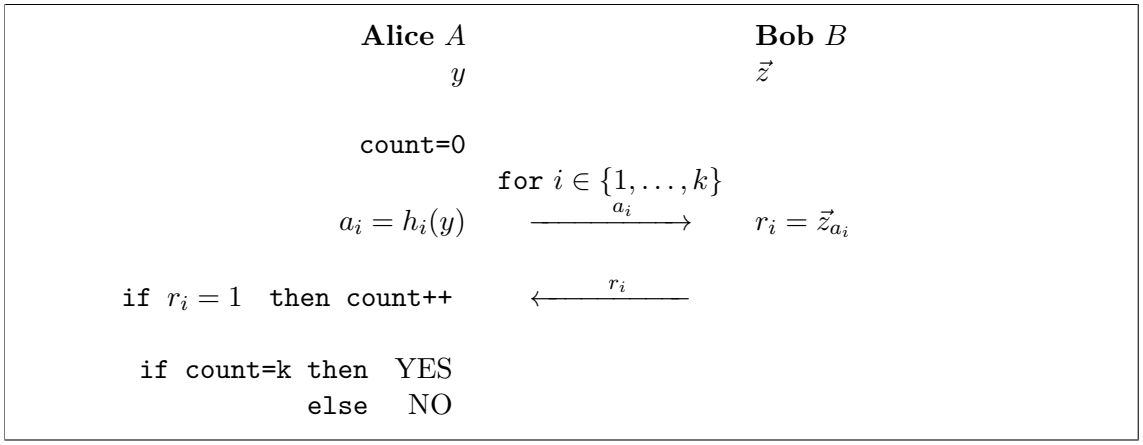

Figure 4.4: Verification on a constraint channel.

$a_i = h_i(y)$, for $1 \leq i \leq k$. In Figure 4.4, we suppose that Alice and Bob first agree on the hash functions to be used. Then Alice sends a_i to Bob. In reply, Bob returns the bit at index a_i of $\vec{z}$ to her. If she only receives 1, y is included in $\vec{z}$ (with a small false positive probability f) otherwise it is not.

The above protocol provides little if no privacy, but can be modified to design a private membership query protocol using PIR as a subroutine. The idea then is to invoke PIR on each query to the filter. The protocol is explained below and requires that the database owner builds a Bloom filter $\vec{z}$ using the entries and a user queries the filter and not the database.

1. Database owner builds the Bloom filter $\vec{z}$ using k hash functions $\{h_1, \ldots, h_k\}$.
2. User for a data y generates $\{h_1(y), \ldots, h_k(y)\}$.
3. For each $1 \leq i \leq k$, the user invokes a single-server PIR on index $h_i(y)$ and retrieves $z_{h_i(y)}$.
4. If $z_{h_i(y)} = 0$ for any i, then y is not in the database, else if all the returned bits are 1, then the data is present (with a false positive probability f).

The only difference with the classical use of Bloom filter (Figure 4.4) in the protocol is that the bit is retrieved using PIR. It is pertinent to note that due to the use of Bloom filters, the protocol entails a false positive probability. However, if it is small of the order 2^{-80} or lower, then we can safely assume it to be negligibly small and hence can be ignored.

Remark 4.6.1 *As in the case of PIR, the database owner in our scenario is honest-but-curious. This attack model for instance does not allow the database owner to return a wrong bit to the user. Under this adversary model, the above protocol modification is private (i.e., keeps user's data private), if the underlying PIR scheme is private. PIR hides any single query of the user from the database owner. Therefore, any k different queries of the user are also hidden by PIR. While this argument is intuitively sound, we however note that a more formal proof would show that a sequential composition of two*

or more PIR instances does not reveal any query index to the database owner. In order to show this, we may develop a proof by reduction to show that if the database owner can learn any query index from two PIR instances, then he can break a stand-alone instance of PIR. We defer this as a future work.

4.6.3 Extension with PBR Protocol

The adapted protocol in its current form requires a bit retrieval PIR scheme. Nevertheless, it can be easily modified to work even with a block retrieval *aka* PBR protocol. The essential advantage of using a PBR protocol instead of a classical PIR protocol would be to increase the throughput, *i.e.*, decrease the number of bits communicated to retrieve 1 bit of information. In fact, the most efficient PIR schemes [GR05], [Lip05] are block retrieval schemes. The modification required to incorporate PBR would consist in using a *Garbled Bloom filter* (see [DCW13]) instead of a Bloom filter. We briefly explain below the garbled Bloom filter construction, and later we present the modification required.

Garbled Bloom Filters. At a high level a Garbled Bloom Filter $(k, m, \mathcal{H}_k, \lambda)$ GBF described in [DCW13] is essentially the same as a Bloom filter. The parameter k denotes the number of hash functions used, while $\mathcal{H}_k$ is a family of k independent hash functions as in a Bloom filter. The size of the filter is denoted by m, and λ is the size of the items to be included in the filter. The difference with respect to a Bloom filter is that at each index in a GBF, a bit string of length λ is stored instead of just storing the bit 1. In order to include an item $y \in \{0,1\}^\lambda$, one randomly generates k shares $\{r_1^y, \ldots, r_k^y\}$, where $r_i^y \in \{0,1\}^\lambda$ such that $y = \oplus_i r_i^y$. As in a Bloom filter, one then generates the k indices $i_1^y, \ldots, i_k^y$ by computing the hashes as $i_j^y = h_j(y)$ and truncating them by taking modulo m. Finally, at index i_j^y of the filter, the bit string r_j^y is stored. Collisions on two values y and y' for a certain hash function h_j are handled by choosing the same r_j for both the values.

To check if a given item is in a GBF, one computes the truncated hashes and retrieves the shares stored at these indices in the GBF. If the XOR of these shares is the same as the given item, then the item is in the filter, or else not. More details on the probability of collisions and the probability of false positives can be found in [DCW13].

Private Membership Query using PBR. This protocol essentially follows the same principle as the one which combines PIR and a Bloom filter. The database owner now builds a GBF $(k, m, \mathcal{H}_k, \lambda)$ using the entries and a user queries the GBF instead of the database. Again k PBR invocations are required to retrieve the k random shares. This adapted protocol is private if the underlying PBR scheme is private, *i.e.*, does not reveal the user's queries. While this argument is intuitively sound, we again note that a more formal proof would show that a sequential composition of two or more PBR instances does not reveal any query index to the database owner. In order to show this, we may follow the same approach discussed for the PIR based membership protocol. The proof may follow a reduction-based argument to show that if the database owner can learn any query index from two PBR instances, then he can break a stand-alone instance of PBR.

We defer this as a future work.

Remark 4.6.2 *At this juncture, we have two solutions for private membership query to public database: 1) k invocations of single server PIR/PBR to a Bloom filter/GBF, 2) Send the complete filter for a local query. On one hand, any PIR based solution only provides computational privacy, has a communication cost, the best being $\mathcal{O}(\log^2 m)$ and involves cryptographic computations and hence entails a significant time complexity. While on the other hand sending the filter ensures absolute privacy, but has a larger communication complexity m bits (still much better than the trivial PIR, i.e., sending the initial database) but has a very low time complexity (has to invoke the protocol in Figure 4.4 locally). Since the size of the database gets drastically reduced with Bloom filter, this solution provides a competitive alternative to trivial PIR even for low memory devices.*

We note that it is possible to further reduce the communication bandwidth of the trivial PIR by sending a Golomb-Compressed Sequence instead of a Bloom filter. This should allow to gain a small factor at the cost of increasing the query time, which becomes linear in case of Golomb-Compressed Sequences. We hence recommend the use of Bloom filters for their constant query time.

4.7 Practicality of the Solutions

We reiterate that a private membership query protocol provides an immediate solution for designing privacy-friendly alerting websites. For the sake of practicality, any realistic privacy-friendly alerting websites should provide response to a user's query in real time. It is hence highly important to evaluate the practicality of the underlying protocol.

We first discuss the practicality of the solutions based on PIR/PBR and Bloom filters in case of public databases and in the sequel we discuss the practicality of PSI/PSI-CA protocol in case of private databases.

Since Bloom filters are highly efficient in space and time, the practicality of PIR/PBR based protocol depends on the practicality of the underlying PIR/PBR scheme. Hence, we first discuss its practicality as perceived in the literature and later by experimentally evaluating PIR/PBR protocols.

For experimental evaluation, the tests were performed on a 64-bit processor desktop computer powered by an Intel Xeon E5410 3520M processor at 2.33 GHz with 6 MB cache, 8 GB RAM and running 3.2.0-58-generic-pae Linux. We have used GCC 4.6.3 with `-O3` optimization flag.

4.7.1 Applicability of PIR

Sion and Carbunar [SC07] evaluate the performance of single database PIR scheme. The authors show that the deployment of non-trivial single server PIR protocols on real hardware of the recent past would have been orders of magnitude less time-efficient than trivially transferring the entire database. The study primarily considers the computational PIR protocol of [KO97]. The authors argue that a PIR is practical if and only if per-bit server side complexity is faster than a bit transfer. With a normal desktop machine, trivial

transfer (at 10 MBps) of the database is 35 times faster than PIR. This ultimately restricts the use of PIR protocols for low bandwidths (tens of KBps).

Olumofin and Goldberg [OG12] refute the general interpretation in [SC07] that no PIR scheme can be more efficient that the trivial one. Authors evaluate two multi-server information-theoretic PIR schemes by Chor *et al.* [CGKS95] and by Goldberg [Gol07] as well as a single-server lattice-based scheme by Aguilar-Melchor and Gaborit [MG08]. The latter scheme is found to be an order of magnitude more efficient over the trivial scheme for situations that are most representative of today's average consumer Internet bandwidth. Specifically, for a database of size 16 GB, the trivial scheme outperforms the lattice based scheme only at speeds above 100 Mbps. More recently, another work by Melchor *et al.* [MBFK16] presents a lattice based PIR which is computationally efficient at the cost of a communication overhead. An implementation of the protocol is also publicly available[6] and is the most efficient (in terms of computation) PIR implementation to this date.

4.7.2 Experimental Analysis

We have tested two PIR/PBR protocols: 1) Cachin *et al.* [CMS99], which is the most efficient (in terms of communication) bit retrieval scheme 2) Melchor *et al.* [MBFK16] which is the most computationally efficient PBR protocol. We have also implemented RSA-OPRF PSI protocol of De Cristofaro *et al.* [DCT10] and PSI-CA protocol of De Cristofaro *et al.* [DCGT12].

4.7.2.1 Public Database

The cost of using PIR-based schemes reduces to the cost of building the filter combined with the cost of k PIR invocations on the filter. We present the time required to build a Bloom filter for the leaked databases corresponding to Snapchat, LinkedIn and Adobe in Table 4.3. The filter is constructed using SHA-1 which generates 20 bytes' digest.

Table 4.3: Results for the leaked databases using SHA-1. Databases contain single data for a user, for instance Snapchat contains only username and ignores other auxiliary leaked information. The experimental results are shown with decreasing false positive probability f.

Database	Size	n	$-\log_2 f$	m (MB)	Build time (mins)	Compress. ratio
Snapchat	49 MB	4609621	32	26	1	1.88
			64	52	2	0.94
			128	102	6	0.48
LinkedIn	259 MB	6458019	32	36	2	7.19
			64	72	3	3.60
			128	142	10	1.82
Adobe	3.3 GB	153004872	32	102	30	33.13
			64	206	72	16.4
			128	412	198	8.20

From Table 4.3, we can observe that the filter size grows slowly and that the compu-

[6] https://github.com/XPIR-team/XPIR

tational time of the filter is reasonable. Initially, all the computations are performed in a sequential manner. We have then distributed the computation on 4 computers (with similar characteristics). Parallelizing the creation of the Bloom filter is straightforward and we nearly achieved a 4× speedup (50 mins). With a few computers, it is possible to reduce the computational time for creating the filter to a desired threshold. We further note that building a Bloom filter involves only a one-time cost.

Despite the space and time efficiency of Bloom filters, the actual bottleneck is the cost of PIR invocation (using the existing primitives). The protocol [CMS99] takes over 6 hours in case of Snapchat database for one invocation. If the probability of false positive is 2^{-32}, *i.e.*, $k \approx 32$, the estimated time for 32 PIR invocations is over 32×6 hours, *i.e.*, over 8 days. The protocol by Melchor *et al.* [MBFK16], however is very efficient and takes only 4 mins. The parameters of the protocol are so chosen such that the security level is of 97 bits. The parameters can be obtained by the optimizer that comes along with the actual implementation of the protocol. Now, let us compare these results with the cost of sending the Bloom filter to the client. Considering the household network bandwidth of 10 Mbps, the time to download the filter would take 20 seconds. The time efficiency of the trivial PIR with Bloom filter seems unmatchable.

4.7.2.2 Private Database

Table 4.4 presents results obtained for the PSI protocol by De Cristofaro *et al.* [DCT10] for 80 bits of security. Our implementation ignores (for the sake of security) some of the optimizations proposed in [DCT12], such as the use of 3 as public RSA exponent and the use of *Chinese Remainder Theorem* for computing RSA signatures. As the user's set has only one data, his computational cost is negligible. To be precise, a user's computational cost consists in computing a signature and n comparisons. The authors in [DCT12] claim that the result of the server's computation over its own set can be re-used in multiple instances. Hence, the server's cost can be seen as a one-time cost, which further makes it highly practical. We present in Table 4.5, results obtained using PSI-CA protocol by De Cristofaro *et al.* [DCGT12]. Recommended parameters of $|p| = 1024$ and $|q| = 160$ bits have been chosen.

Table 4.4: Cost for PSI protocol [DCT10] with 80 bits of security using SHA-1.

Database	Cost (mins)
Snapchat	48
LinkedIn	68
Adobe	1600

Table 4.5: Cost for PSI-CA protocol [DCGT12] with 80 bits of security using SHA-1.

Database	Cost (mins)
Snapchat	9
LinkedIn	12
Adobe	301

Clearly, PSI-CA outperforms PSI by a factor of 5. The reason behind this performance leap is that the exponents in modular exponentiations are only 160 bits long in PSI-CA as opposed to 1024 bits in PSI.

Table 4.6 summarizes the results obtained on Snapchat database for $f = 2^{-32}$. Clearly,

Table 4.6: Summary of the results on Snapchat with $f = 2^{-32}$.

		Cost	
Protocol	**Type**	**Commun.**	**Comput.**
Trivial PIR with Bloom filter	PIR	26 MB	1 min
Cachin *et al.* [CMS99]	PIR	7.8 KB	$>$ 8 days
Melchor *et al.* [MBFK16]	PBR	3 TB	4 mins
De Cristofaro *et al.* [DCT10]	PSI	562 MB	48 mins
De Cristofaro *et al.* [DCGT12]	PSI-CA	87.92 MB	9 mins

in the public database case, sending the Bloom filter is the most computationally efficient solution. While, in the private database scenario, PSI-CA provides a promising solution. Comparing the two cases, we observe that the private database slows down the query time by a factor of 9.

We highlight that PSI/PSI-CA protocols perform much better than PIR/PBR protocols. This is counter-intuitive, as in case of PIR the database is public while in PSI the database is private. A protocol on private data should cost more than the one on public data. From a theoretical stand-point, there are two reasons why private set intersection protocols perform better than PIR protocols: 1) the computational cost in PSI/PSI-CA protocols is reduced at the cost of communication overhead, 2) the size of the security parameter is independent of the size of the database. More precisely, the communication cost of the most efficient PSI/PSI-CA protocols [DCT10, DCGT12], [HEK12, DCW13] is linear while the goal of PIR protocols is to achieve sub-linear or poly-logarithmic complexity. This indeed comes at a cost, for instance the size of RSA modulus in PSI [DCT10] for 80 bits of security is 1024 bits and hence independent of the size of the sets involved. While in case of PIR [CMS99], the size of the modulus used is $\log^{3-o(1)}(n)$ bits. Hence for a million bit database, the modulus to be considered is around 8000 bits, which leads to a very high computational cost. The protocol in [MBFK16] is computationally efficient at the cost of a high communication overhead which is essentially due to the long ciphertexts used in the underlying encryption scheme.

4.8 Related Work

Nojima *et al.* [NK09] present a protocol to privately query a Bloom filter. They consider the private database case, hence the Bloom filter must remain private to the server, while a client should be able to privately query for an item. The proposed protocol is based on blind RSA signatures [Cha82]. The idea is the following, the server builds a Bloom filter that contains the signatures generated on the items of the database. This filter is then sent to the client. Whenever, the client wishes to test the presence of an item, he asks for a blind signature on the item and then can locally perform a query to the Bloom filter. The computation cost is the cost of generating the n signatures (by the server), where n is the size of the database and the computation of a blind signature (by the client and the server). Additionally, there is the cost of building the filter. Since, the filter is sent to the client in clear, the protocol leaks the positions of 1 in the filter. Revealing the number of

1s in the filter reveals the size of the initial database. A strengthened protocol requires building an encrypted filter. This construction is presented in [MLRN15]. The paper also evaluates this strengthened protocol along with some other privacy-preserving protocols to query the filter. However, the protocol is shown to take over 27 hours to generate the encrypted Bloom filter. The result of evaluation of other protocols is not very promising either.

A more recent work by Tamrakar *et al.* [TLP+16] presents a private membership query protocol using a trusted hardware component. The authors propose a *carousel design pattern* in which the representation of the database (such as a Bloom filter) is continuously circled through the trusted hardware on the server. Clients can then communicate with the trusted hardware via secure channels.

4.9 Summary

In this chapter, we studied the privacy provisions of Bloom filters through several use cases. We first showed why the naive use of Bloom filters gives a false sense of privacy. Next, we provided an overview of two state-of-the-art tools for private computation. Third, we examined an application of Bloom filters in the context of websites alerting users about data leakage.

With the current rate of leakage, these alerting websites will be needed for a while. Unfortunately, it is currently difficult to determine whether or not these websites are phishing sites since they do not provide any privacy guarantee to users. The analysis presented here exposes the privacy risks associated to the most popular alerting websites. We further evaluate how state-of-the-art cryptographic primitives can be applied to make private queries to an alerting site possible. Two different scenarios have been considered depending on whether the database is public. While PSI/PSI-CA protocols provide a straightforward solution in the private database scenario, a tweak using Bloom filters transforms PIR/PBR into private membership protocols for public database.

Our experimental evaluation shows that PSI/PSI-CA protocols perform much better than PIR/PBR based protocols. This is an encouraging result for the ethical hacking community or security companies. Using private set-intersection protocols, an ethical hacker may now privately inform a user about his leaked data without revealing any information on other users.

Chapter 5

Security of Bloom Filters

Contents

5.1 Introduction **70**

5.2 Adversary Models **71**

5.2.1 Chosen-insertion Adversary 73

5.2.2 Query-only Adversary 75

5.2.3 Deletion Adversary 77

5.2.4 Summary 77

5.3 Application Level Tools **78**

5.3.1 Web Spider: Scrapy 79

5.3.2 Bitly Spam Filter: Dablooms 82

5.4 Networking Tools **83**

5.4.1 Web Proxy: Squid 83

5.4.2 Intrusion Detection System: AIEngine 84

5.5 Forensic Tools **85**

5.5.1 NSRL Forensic Filter 85

5.5.2 Extension to sdhash 86

5.6 Predicting Unknown Filter Parameters **87**

5.6.1 Number of Hash functions 87

5.6.2 Filter Size 88

5.6.3 Hash Functions 89

5.7 Countermeasures **90**

5.7.1 Alternate Data Structures 90

5.7.2 Worst-case Parameters for Bloom Filters 94

5.7.3 Probabilistic Solutions 95

5.8 Related Work **97**

5.8.1 Attacks 97

5.8.2 Defense 98

5.9 Summary **100**

5.1 Introduction

Owing to their succinct representation of large sets, Bloom filters have gained huge popularity among software developers. To give a sense of their widespread use, GitHub includes over 700 repositories that either incorporate or implement Bloom filters in over 10 programming languages. The data structure is even more appealing to the research community as a query to Google scholar reveals (almost literally) that a myriad of papers have used Bloom filters or their extensions in a variety of largely diverse scenarios.

With the purpose of ensuring a robust service, Bloom filters have particularly been employed in sensitive security infrastructures such as forensic tools [Nat16, Rou10], intrusion detection systems [AIE16] and spam filters [Bit12], among many others. This chapter focuses on such sensitive infrastructures and raises the following important questions:

> *Does the inclusion of Bloom filters increase the attack surface of a system? How difficult it is for an adversary to exploit Bloom filters to mount attacks?*

This chapter addresses this question by presenting a two-dimensional analysis of Bloom filters. First, we define multiple adversary models for Bloom filters in a stand-alone setting. The adversary models categorically represent the different access privileges that an adversary can obtain and the expected impact that she can exert. Second, we put our adversary models to test on six sensitive infrastructures to demonstrate their real-impact.

Bloom filters apparently come with two inherent weaknesses: 1) They are probabilistic data structures, a property that an adversary can exploit so that a filter systematically shows the worst-case behavior. 2) Even more importantly, Bloom filters are hash-based data structures. Whence, it does not come as a surprise that an adversary may rely on the vulnerability of weak hash functions or the inordinate use of secure hash functions. Indeed, in the past, similar vulnerabilities in hash tables have been exploited by Crosby and Wallach [CW03]. In fact, a careful scrutiny of Bloom filter enabled tools reveals that the best practices on the usage of hash functions (see [Dan12]) are rarely respected. Developers are tempted to employ non-cryptographic hash functions to boost the performance of their applications. Another temptation for developers is digest truncation. Hash functions, in general, produce more bits than required by a filter. Several bits of the digest are thus discarded leading to potential weaknesses.

The adversary models presented in this chapter essentially define the manner in which these weakness can be exploited. The main result of our adversarial models is the computation of the worst-case error probability for Bloom filters. We show that if Bloom filter parameters are not chosen according to the worst-case error probability, an adversary can pollute a filter with well chosen inputs, forcing it to deviate from its normal behavior. She can also query the filter to make it produce erroneous answers (or false positives). Once again, our attacks rely on a kind of forgery similar to finding pre-images and second pre-images of digests.

We make the following contributions in this chapter:

1. We present adversary models for Bloom filters. We show how they can be pol-

luted/saturated using pre-image attacks and how this increases the false positive probability. Then, we show how to forge false positives.

2. To validate our adversary models, we demonstrate attacks on 6 software applications: Bloom-enabled Scrapy web spider, Bitly Dablooms spam filter, Squid web cache, forensic tools: NSRL Filter and `sdhash`, and an intrusion detection system called AIEngine.

3. Our adversary models assumes that the filter parameters are known to the adversary. Another orthogonal contribution in this regard is to modify this assumption and to study how an adversary can determine these parameters when only given an oracle-access to the filter. The obtained results extend our attacks to proprietary Bloom filter implementations or to filters stored on distant servers.

4. In the adversarial settings, we have the liberty to assume that the inputs to the filter are non-uniformly distributed. This observation leads to our third contribution: we compute the worst-case false positive probability and obtain new equations for Bloom filter parameters. These equations yield new parameters that reduce the adversary's advantage. We also show how to strengthen Bloom filters without changing the parameters, and provide techniques to save calls to cryptographic hash functions. The trade-offs between the query time, the memory consumption and the security of the countermeasures are also explored.

5. We also compare a data structure due to Bloom [Blo70] as a countermeasure. *Bloom Hash Tables*, as we refer to them are more appealing than Bloom filters as they often require less memory and additionally they resist better to our attacks.

This chapter is divided into four parts. In the first part, we describe our adversary models (Section 5.2). In the second part, we present the illustrations: Section 5.3 presents application level tools that include Dablooms and Scrapy; Section 5.4 presents networking tools that include Squid and AIEngine; Section 5.5 presents forensic tools that include NSRL filter and `sdhash`. In the third part, we describe the strategies that an adversary can employ to learn the filter parameters (Section 5.6). The fourth part of this chapter describes countermeasures (Section 5.7) and the related work (Section 5.8). We note that throughout this chapter we employ the notations developed in Chapter 2, §2.4.

5.2 Adversary Models

As a general principle, developers while designing applications built on a Bloom filter decide on the maximum number of elements to be inserted, the desired false positive probability and a hash function. Once these parameters are chosen, they can compute the filter size and the optimal number of hash functions using (2.2) and (2.3).

In this work, we assume that Bloom filters are always deployed and maintained by trusted parties. This assumption is necessary, otherwise any bit of a filter can be tampered by the adversary. The scenario where Bloom filters are maintained by possibly untrusted

parties is hence meaningless and has indeed led to big failures such as List Of All Friends (LOAF)[1]. LOAF (now discontinued) was designed to allow a user to send email messages along with his address book compressed in the form of a Bloom filter. The motivation behind sending address books was that the friends of my friends are trusted. Therefore, the Bloom filters of a user's friends can be used as a *whitelist* spam filter. When an email is received, the source address is checked against the Bloom filter. If it is present, the email is not marked as spam. Otherwise, it is suspicious and must be analyzed using a more complex spam filter. The trivial attack here is to send a fake Bloom filter (for instance, $\vec{z}$ where all the bits are set to 1) allowing a malicious user to whitelist any email address in the world.

We also assume that the implementation of the Bloom filter is public and known to the adversary. Moreover, we assume that the operations on the filter are always predictable. These hypotheses are usually verified in open source software.

Before describing our adversary models, we define *Pre-image* and *Second pre-image attacks* on Bloom filters. These attacks can be perceived as natural extensions of their hash function counterparts. We recall that a hash function producing ℓ bits of digest entails pre-image and second pre-image attacks with complexity 2^{ℓ}, *i.e.*, with a success probability of $1/2^{\ell}$ (See Chapter 2, §2.1). The corresponding concepts for Bloom filters however require explicit treatment due to the subtleties introduced by the data structure.

Definition 1 ***Pre-image for a Bloom filter.*** *Given a Bloom filter, $\vec{z}$, with only one item inserted into it, an associated pre-image for the filter is a string $y \in \{0,1\}^*$ with $I_y = \{h_1(y), \ldots, h_k(y)\}$ such that $I_y \subseteq supp(\vec{z})$.*

Definition 2 ***Second pre-image for a Bloom filter.*** *Given a Bloom filter, $\vec{z}$, with only one item $x \in \{0,1\}^*$ with $I_x = \{h_1(x), \ldots, h_k(x)\}$ inserted into it, an associated second pre-image for the filter is another string $y \neq x$ with $I_y = \{h_1(y), \ldots, h_k(y)\}$ such that $I_y \subseteq supp(\vec{z})$. The difference with respect to pre-images is that the item x is now given to the adversary.*

Remark 5.2.1 *Thinking along the lines of (second) pre-image attacks on hash functions, the complexity of finding a pre-image or a second pre-image for a Bloom filter should intuitively be $\frac{1}{m^k}$, for each hash function h_i is uniform over $[0, m-1]$. However, since in case of Bloom filters, the order of hashes in the set I_y is not important, the effective probability of finding a (second) pre-image in case of Bloom filters is $\frac{w_H(\vec{z})^k}{m^k} = \left(\frac{w_H(\vec{z})}{m}\right)^k$. This holds due to the fact that the total number of (second) pre-images is $w_H(\vec{z})^k$, i.e., all permutations of $supp(\vec{z})$ with repetitions.*

We further note that if in the above definitions, we only assume a non-empty filter instead of a filter containing a single item, then (second) pre-images for the filter correspond to "special" false positives. These false positives are special as they share the same bit positions as that of an item already present in the filter. This is best explained in Figure 5.1, where both y_1 and y_2 are false positives, but only y_2 defines a second pre-image of x_2.

[1] https://wiki.apache.org/spamassassin/Loaf

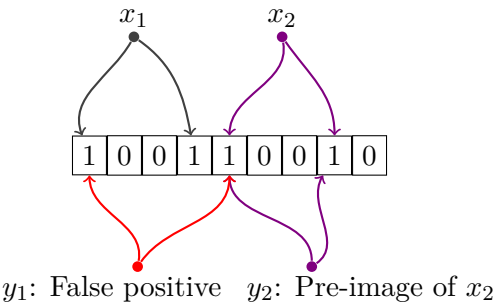

Figure 5.1: Items x_1 and x_2 are inserted into the filter using 2 hash functions. While both y_1 and y_2 are false positives. Only y_2 defines a pre-image of x_2. This is because the first digest of y_1 corresponds to a bit occupied by x_1, while the second digest corresponds to a bit occupied by x_2.

In the following, we define three adversaries for Bloom filters: *chosen-insertion adversary*, *query-only adversary* and *deletion adversary*.

5.2.1 Chosen-insertion Adversary

The first adversary can choose the items to be inserted in the Bloom filter. She can either add the items to the filter by herself or can arrange to make the trusted party do it for her. We consider two cases: in the first scenario the adversary controls all the items to be inserted in the filter and the second in which the filter is non-empty at the time of the attack and hence contains some already inserted items. The goal of this adversary in both cases is to obtain a false positive probability which is higher than the one expected by the developer. This is achieved by increasing the number of set bits in the filter, which we call a *pollution attack*. In the worst case, the adversary can set all the bits of the filter to one: this is called a *saturation attack*.

By carefully choosing the items, it becomes possible to exceed the expected false positive probability, which eventually forces the application to deviate from its expected behavior. To be precise, a *polluting item* x should maximize the number of bits set to 1:

$$\begin{aligned} \forall i \neq j \in [1,k],\ h_i(x) \neq h_j(x)\ , \\ \forall i \in [1,k],\ h_i(x) \notin \mathrm{supp}(\vec{z})\ . \end{aligned} \tag{5.1}$$

In the above equation, $\vec{z}$ denotes the filter after each insertion. After n such insertions into the filter, the number of set bits attains the value kn.

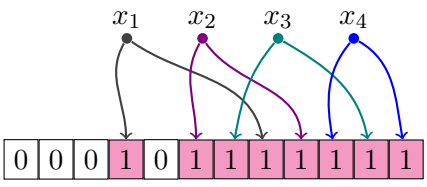

Figure 5.2: Chosen-insertion adversary ($k = 2$).

An illustrative example is presented in Figure 5.2. Items x_1, x_2, x_3, x_4 are so chosen

such that all $h_j(x_i)$ are distinct, for $j \in [1,2]$ and $i \in [1,4]$. The colored cells are the bits set to 1 by the adversary after crafting the corresponding items.

Hence, with carefully selected items, the number of set bits in the filter can be made larger than the expected value. In fact, as previously mentioned (see Inequality (2.5)), with the optimal parameters: m, n, and k_{opt}, the expected number of set bits in the filter is (refer to (2.2)):

$$\begin{aligned}\frac{m}{2} &= \frac{nk_{\text{opt}}}{2\ln 2} \\ &\approx 0.72nk_{\text{opt}} \quad .\end{aligned}$$

Comparing it to nk_{opt} bits set to 1 by the adversary, she increases the number of 1s in the filter by 38%. For a chosen k, the adversary sets nk bits of the m-bit filter to 1, hence the false positive probability achieved in the attack is:

$$f^{\text{ADV}} = \left(\frac{nk}{m}\right)^k \quad . \tag{5.2}$$

Figure 5.3 shows how the false positive probability behaves under a chosen-insertion attack. We choose a Bloom filter of size $m = 3200$ with a capacity of 600 items. Equation (2.2) for optimal parameters gives $k_{\text{opt}} \approx 4$, and $f_{\text{opt}} = 0.077$. When the number of inserted items is low, f^{ADV} and f are superimposed until $\lceil \frac{\sqrt{m}}{k} \rceil$ items have been added. This is due to the *Birthday paradox*: the first $\sqrt{m}$ items' indexes are likely to be all different. It implies that the adversary does not need to compute pre-images for the first $\lceil \frac{\sqrt{m}}{k} \rceil$ items she wants to insert. Let us now consider the threshold probability of $f_{\text{opt}} = 0.077$. This threshold is reached after 600 insertions if the indexes are uniformly distributed. An adversary however can attain this value after only 422 well-chosen insertions. After 600 chosen insertions, she obtains a false positive probability of $0.314 \approx 4f_{\text{opt}}$. In certain scenarios, an adversary may not be able to control all the insertions into the filter. To this end, let us consider the case of 400 normal insertions followed by insertions chosen by the adversary. The threshold of 0.077 is then reached after 510 insertions. At the end of 600 insertions, she obtains a false positive probability of $0.17 \approx 2f_{\text{opt}}$.

Now, we consider saturation attacks. We first note that a filter with optimal parameters cannot be saturated. To see this, it is clear that a Bloom filter can be saturated only when all the m bits of the filter are set to 1. This can be achieved by carefully choosing items to be inserted into the filter such that each item sets a maximum of k bits to 1. Hence, after n insertions, the number of bits set to 1 in the filter can at most be nk. Since the filter parameters are optimal, we have:

$$nk = m\ln(2) = 0.693m < m.$$

This shows that the adversary needs to insert more than the optimal number of items in the filter to saturate it. We now consider the scenario where one can insert more items than the optimal number. To this end, let us first consider the benign case, where only random items are inserted. In this case, the expected number of items required to fully

saturate a filter is: $\Theta(\frac{m \log m}{k})$. This directly follows from the coupon collector's problem (see Chapter 3 §3.2.2). In the case of a Bloom filter, k coupons are drawn in each draw. An adversary can however improve this number. In contrast to the benign case, an adversary who inserts carefully chosen items only requires

$$\left\lceil \frac{m}{k} \right\rceil \text{ items,} \tag{5.3}$$

since each item sets k bits to 1 and there are m in total. This allows the adversary to gain a factor of $\Theta(\log m)$ to achieve saturation.

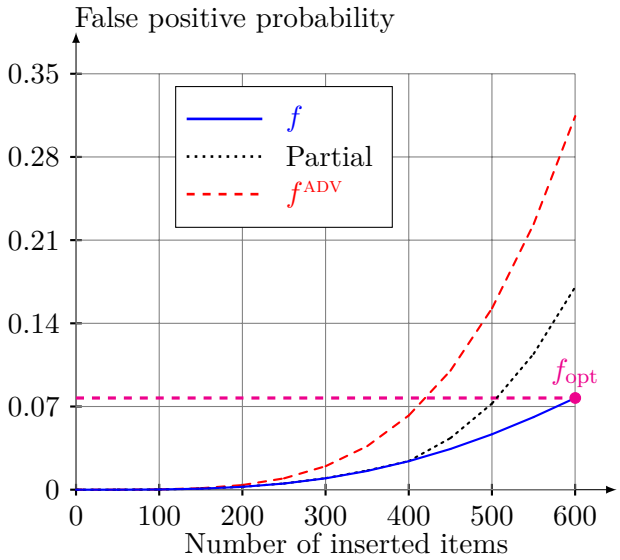

Figure 5.3: False positive probability as a function of inserted items ($m = 3200$, $k = 4$ and $f_{\text{opt}} = 0.077$).

An important question is to estimate the feasibility of forging a polluting item that satisfies conditions (5.1). To this end, let us consider a filter $\vec{z}$ of Hamming weight $w_H(\vec{z}) = W$, for an integer $W > 0$. We want to insert a polluting item x to $\vec{z}$. There are $\binom{m-W}{k}$ ways to choose such an item. The probability to find such an item is then:

$$\frac{\binom{m-W}{k}}{m^k} \quad .$$

If we wish to insert n polluting items in an empty filter, there are $\binom{m-(n-1)k}{k}$ ways to choose the n-th item. If m is large compared to k, finding polluting elements is much simpler than finding pre-images or second pre-images for a Bloom filter. We illustrate in Section 5.3 with a web spider, the practical cost of forging polluting items.

5.2.2 Query-only Adversary

The second adversary cannot insert items into the filter. However, she knows the current state of the filter or a part of it. Similar to a chosen-insertion adversary, she can either generate queries by herself or force the trusted party to query on her behalf. The query-only adversary can have two distinct objectives. She can either craft items that

generate false positives hence force the application to err, or that her items' digests are well-distributed in the filter leading to latency. We note that these attacks trivially extend to both Counting and Scalable Bloom filters.

There could be several motivations for an adversary to make the filter generate false positives. A typical scenario would be to attack applications incorporating Bloom filters, but which do not tolerate false positives at all. In such applications, Bloom filters conjointly work with a remote mechanism (for example a database storing the items) to get rid of false positives. In general, when the item is found to be present in the filter, the remote mechanism provides a confirmation of a true positive. This ensures that the application does not err. Generation of large number of false positives would lead to *false positive flooding* enabling an adversary to hit the second mechanism and attempt to mount a DoS.

For a given Bloom filter $\vec{z}$, a query-only adversary wants to generate an item y such that:

$$\forall i \in [1,k],\ h_i(y) \in \text{supp}(\vec{z})\ . \tag{5.4}$$

Figure 5.4 shows different examples of false positives: Items y_1, y_2, y_3 are detected as being present in the filter while these items were never inserted in the filter. Items y_1 and y_3 are particularly interesting. While y_1 verifies $h_1(y_1) = h_2(y_1)$, y_3 satisfies $h_1(y_3) = h_2(x_1)$ and $h_2(y_3) = h_1(x_1)$.

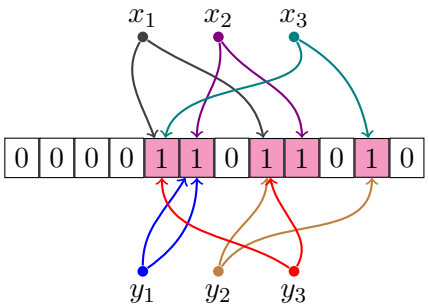

Figure 5.4: Crafted false positives: $w_H(\vec{z}) = 5$. The first digest $h_1(\cdot)$ corresponds to the left arrow for each item, while the second digest $h_2(\cdot)$ corresponds to the right arrow.

Knowing the positions of the 1s, the adversary has $W^k = (w_H(\vec{z}))^k$ choices to forge a false positive. There are 25 choices in Figure 5.4. The probability to forge a false positive y is: $\left(\frac{W}{m}\right)^k$.

It is also possible to consider a query-only adversary making *dummy queries*. The adversary queries for items which are not in the filter. Let y be an item chosen by the adversary, then it must satisfy:

$$\forall i \neq j \in [1,k],\ h_i(y) \neq h_j(y)\ ,$$
$$\forall i \in [1,k-1]\ , h_i(y) \in \text{supp}(\vec{z}) \text{ and } h_k(y) \notin \text{supp}(\vec{z})\ .$$

The probability of finding such an item is:

$$\frac{(m-W)\cdot\binom{W}{k-1}}{m^k}\ .$$

The idea of this attack is to make the query as expensive as possible. It targets applications with very large Bloom filters and forces the party running the Bloom filter to make more memory accesses and more computations than expected. The goal of the adversary is to reach the worst case execution time for each query.

5.2.3 Deletion Adversary

This adversary targets Counting Bloom filters. It is well known that *false negatives* are the drawbacks of these variants [GLLY10]. Hence, an adversary who does not control the insertions into the filter can nevertheless forge an item and make it delete from the filter.

We assume that the filter is fully or at least partially known to the deletion adversary. The goal of the adversary is then to create false negatives: she wants to make an item x with $I_x = \{h_1(x), \ldots, h_k(x)\}$ disappear from the filter. To this end, the adversary needs to find item(s) x' in the filter with $I_{x'} = \{h_1(x'), \ldots, h_k(x')\}$ such that $I_{x'} \cap I_x \neq \phi$. Consequently, deleting the item x' from the filter would decrease at least one counter for x and hence the item x also gets deleted. The probability to find such an x' is given by:

$$1 - \left(\frac{W - |I_x|}{m}\right)^k, \text{where } |I_x| \text{ is the cardinality of } I_x.$$

We highlight that deletion of an item may require other deletions. These deletions may remove several other items from the filter as a side effect.

5.2.4 Summary

To summarize, our attacks against Bloom filters can be classified depending on the bit of the filter targeted by the adversary. The chosen-insertion adversary generates pre-images on the 0 of a Bloom filter. The query-only adversary generates pre-images on the 1 of a Bloom filter and sends queries for which the bits at the first $k-1$ indexes are set to 1, while the last index could be either 0 or 1. The deletion adversary targets certain 1s in the filter. In each case, we consider brute force search: an item is selected at random and its k indexes are computed. If the bit in the filter at any of these indexes is already set to 1 or 0 depending on the adversary, the item is discarded and a new one is tried.

Feasibility of our attacks. We highlight that our attacks on Bloom filter based applications are feasible since either non-cryptographic hash functions are used or cryptographic digests are truncated. While non-cryptographic hashes can be easily broken, truncation of cryptographic digests drastically reduces the attack complexity. Furthermore, the probability of success of our attacks is higher than the generally accepted (second) pre-image attacks on hash functions. Table 5.1 presents the summary of the success probability of our attacks.

From the computed probabilities (see Table 5.1), it is possible to order each attack in terms of their computational feasibility. The pollution attack has the highest success probability. The most difficult attack is the deletion one. Between these two attacks, lies

Table 5.1: Comparative summary of our attacks for a filter of Hamming weight W. ℓ is the size of the digest generated by a cryptographic hash function.

Attack	Probability
(Second) pre-image (hash function)	$\frac{1}{2^\ell}$
(Second) pre-image (Bloom filter)	$\left(\frac{W}{m}\right)^k \leq \left(\frac{k}{m}\right)^k$
Pollution	$\frac{\binom{m-W}{k}}{m^k}$
False positive forgery	$\left(\frac{W}{m}\right)^k \leq \left(\frac{1}{2}\right)^k$
Dummy query	$\frac{(m-W)\cdot\binom{W}{k-1}}{m^k}$
Deletion	$1-\left(\frac{W-k}{m}\right)^k \leq 1-\left(\frac{W-\lvert I_x\rvert}{m}\right)^k$

the difficulty of mounting query attacks which depends on the number of items inserted in the filter.

We now put our adversary models to test on the following three classes of software applications:

1. **Application level tools:** These include Scrapy web browser [Scr16] and Dablooms spam filter [Bit12]. In case of Scrapy, we show a chosen-insertion attack and a query-only attack, while for Dablooms, we show a chosen-insertion attack and a deletion attack.

2. **Networking tools:** These include Squid [Wes04]—a web proxy and AIEngine which is a network intrusion detection system. For Squid, we first show a chosen-insertion attack and then we build a query-only attack on top of it. As for AIEngine, we show a query-only attack.

3. **Forensic tools:** These include the NSRL forensic filter [Nat16] and `sdhash` [Rou10], a similarity based hashing mechanism that extends NSRL filter. We show query-only attacks to bypass these forensic tools.

We note that all the tests presented in this chapter except the one in Section 5.6 (for which we later provide the characteristics) were performed on a 64-bit processor laptop computer powered by an `Intel Core i7-4600U CPU` at `2.10GHz` with `4MB cache`, `8GB RAM` and running `Linux 3.13.0-36-generic`.

5.3 Application Level Tools

In the following, we consider two applications namely, Scrapy, a web spider that allows the usage of Bloom filters to implement a *déjà vu* URL list, and Dablooms, a malware filter designed by Bitly to detect malicious URLs.

5.3.1 Web Spider: Scrapy

A *spider*, also known as a *robot* or a *crawler* is a mechanism for bulk discovering or downloading of publicly available web pages from the world wide web. Web crawler forms an important component in the design of web search engines: they assemble a corpus of web pages, index them and allow users to issue queries for a keyword and retrieve matching pages. Another closely related use case is web archiving, where a large set of web pages is periodically collected and archived for posterity. They are also useful in web data mining, where web pages are analyzed for statistics or are monitored for copyright infringements [Att16]. Recently, crawlers have been appositely employed to design automatic tools to track the web for personalized topics and notify clients of pages that match these topics [Gig16]. Built as an automated tool, these regular notifications effortlessly keep clients informed of the changes on the web which they can not afford to be oblivious of. Interested readers may consult the survey by Olston and Najork [ON10] for more applications and further detail.

In this section, we first present Scrapy [Scr16], a high level screen scraping and web crawling framework, used to crawl websites and extract structured data from their pages. It has been employed for a wide range of applications, varying from data mining, monitoring, to information processing and automated testing, among others[2]. In the sequel, we present attacks against Scrapy.

5.3.1.1 Scrapy: A Simplified Architecture

Scrapy extracts data from HTML and XML sources with the help of XPath and provides built-in support for sanitizing the scraped data using a collection of reusable filters shared between the spiders. Scrapy is capable of performing crawling at different scales and depths. For instance, it can crawl specific domains or pages related to a given topic of interest.

Execution of a crawling process recursively performs the following steps. These steps are common to any spider and hence are not specific to Scrapy.

1. Select a URL from the list of scheduled ones.
2. Fetch the URL.
3. Archive the results.
4. Select URLs of interest and add to the scheduling list.
5. Mark the current URL as visited.

Step (5) is crucial to eliminate previously visited URLs. In light of this, different data structures have been deployed, varying from a list, to hash tables and Bloom filters among others. In case of Scrapy, the default duplicate filter to mark visited URLs uses URL fingerprints of length 77 bytes in Python 2.7. Hence to scrape a site with 2 million pages,

[2] http://scrapy.org/companies/

the list of visited URLs might grow up to 2M × 77B = 154 MB. In order to scrape 100 such domains, a memory of 15 GB would be required. The developers have left the possibility to use alternative data structures open. Considering the low memory footprint of Bloom filters, developers have proposed to employ them in Scrapy[3]. Bloom filters are indeed used in various other crawlers, for instance Heritrix [Her16]. However, we show in the sequel that Bloom filters must be judiciously used, else they might invite serious attacks.

5.3.1.2 Attacks

In this section, we assume that a Bloom filter is used in Scrapy to mark visited URLs. The implementation is based on the Python module pyBloom[4]. It is a popular implementation of Bloom filter in Python and can be easily plugged into Scrapy (version 0.24). pyBloom proposes the following cryptographic hash functions: SHA-512, SHA-384, SHA-256 and MD5. The indexes corresponding to a URL are generated by a hash function with sufficiently large digest and seeded using a known deterministic salt. In turn, the choice of the hash function depends on the size of the filter.

An adversary may easily mount a chosen-insertion attack to pollute the spider's filter. She generates or owns an initial web page. She then creates links on it with well-chosen URLs, such that each URL upon insertion in the filter sets previously unset bits to 1. Whence, the false positive probability of the filter increases to (5.2), if her web page is the starting point of the crawling process. Once the initial web page is completely crawled, Scrapy upon starting to crawl any other page not owned by the adversary would detect it to have been already visited with probability (5.2). Hence, it would terminate the crawling process while falsely believing that the web page has already been crawled — *We have blinded the spider.* We note that this attack is not specific to Scrapy, but extends to any spider employing Bloom filter in an insecure manner.

We performed empirical tests to determine the cost of finding polluting URLs. Our test program is written in Python and the target platform runs CPython 2.7.3 interpreter. We have also employed fake-factory[5] (version 0.2) a Python package to generate fake but human readable URLs.

Figure 5.5 describes the time needed to forge URLs to pollute Scrapy's filter. We choose the number of items $n = 10^6$ and the false-positive probability: 2^{-5}, 2^{-10}, 2^{-15} and 2^{-20}. The size m, k and the hash functions are automatically computed by Pybloom. We observe that the time needed to find the polluting items grows exponentially. This is a direct consequence of the probability of finding a polluting item (see Table 5.1). Indeed, the number of ways to choose the n-th polluting item decreases exponentially as n increases, which explains the exponential increase in time to find these polluting items. For $f = 2^{-5}$, it takes 38 seconds to generate 10^6 URLs, while for $f = 2^{-20}$, it takes 2 hours.

It is also possible to mount query-only attacks. To this end, we assume that the adversary does not want to have some of her pages to be crawled by Scrapy. She does

[3] http://alexeyvishnevsky.com/?p=26
[4] https://github.com/jaybaird/python-bloomfilter
[5] https://pypi.python.org/pypi/fake-factory/0.2

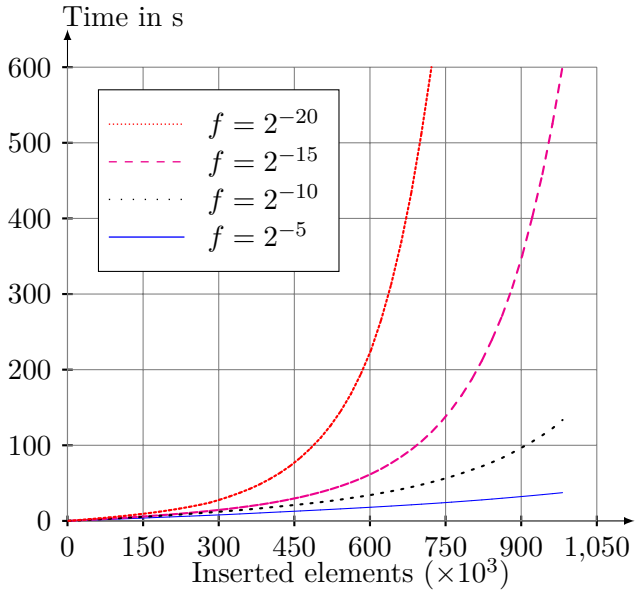

Figure 5.5: Cost of creating polluting URLs.

not trust the `robots.txt` file since many spiders are impolite, *i.e.*, they do not respect `robots.txt` policies. To hide her secret pages from Scrapy, she publishes a few pages (the decoys) with some links to her secret pages (the ghosts). She chooses the URLs of her ghost pages to make Scrapy think that they have already been seen (false positives). All her pages are organized in a web tree such that the leaves are the ghosts, the root (entry point) and all the intermediate nodes are decoys. In the most simple setting, we have one ghost and a single decoy. Given a URL, finding a decoy (or the ghost) has the probability $\left(\frac{k}{m}\right)^k$. This task is time consuming as shown in Figure 5.6.

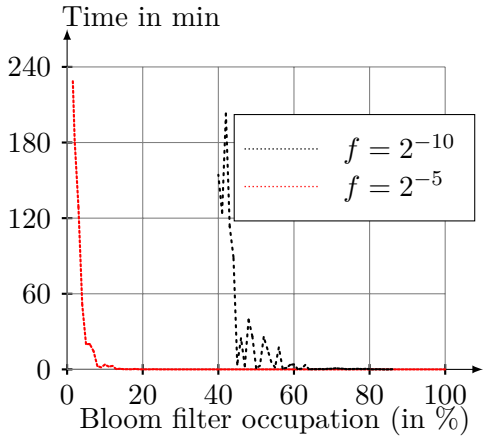

Figure 5.6: Cost of creating ghost URLs. The curve for $f = 2^{-5}$ is truncated as it took several hours to find a false positive when the filter occupation was less that 40%.

Figure 5.6 presents the cost to generate a false positive as a function of the filter occupation (the ratio between the current number of insertions to the capacity of the filter, which is 10^6 in our case). In order to generate false positives, random items were generated and tested against the filter.

A more simple solution consists in generating from a URL several decoys to hide the

ghost. We need $\Theta(k \log k)$ random values to hide a URL (coupon collector's problem). Figure 5.7 illustrates an example for $k = 3$.

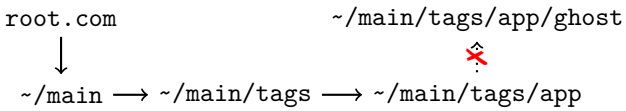

Figure 5.7: A root domain `root.com` with the page tree `main, main/tags, main/tags/app` is chosen. Once SCRAPY recursively visits the decoys, the ghost page `main/tags/app/ghost` is considered as already visited. The ghost page thus remains hidden (hence a dotted arrow and a cross).

5.3.2 Bitly Spam Filter: Dablooms

URL shortening services such as BITLY are targeted by cybercriminals to lure users into phishing/malware traps (see [NMS+14]). In order to prevent misuse, these services apply filters to detect malicious URLs. The current filtering policies were studied in [GAK14]. DABLOOMS is an experimental data structure proposed by BITLY to prevent the shortening of malicious URLs.

DABLOOMS is the combination of two variants of Bloom filters, namely, Counting Bloom filters and Scalable Bloom filters (refer to Chapter 2 §2.4). For the Counting Bloom filter, DABLOOMS uses 4-bit counters. As for the Scalable Bloom filter in DABLOOMS, the parameter r is equal to 0.9. The hash function used in DABLOOMS is MurmurHash [App10] combined with a trick from Kirsch and Mitzenmacher [KM08] to reduce the number of calls to MurmurHash.

We describe the effect of a pollution attack carried out by a chosen-insertion adversary. In the sequel, we also present a deletion attack.

As described earlier, the adversary first generates phishing or malware websites with well-chosen URLs. Then, she needs to make BITLY include her URLs in DABLOOMS. The most obvious choice to achieve this is to flood the web with her malicious URLs and wait until it is spotted by BITLY. Another option is to register her URLs directly to anti-phishing websites such as PHISHTANK[6] to get her URLs recognized as the ones hosting malicious content and to eventually get included in DABLOOMS.

Figure 5.8 describes the effect of pollution on DABLOOMS. There are $\lambda = 10$ filters and each filter can include $\delta = 10000$ items and we have chosen $f_0 = 0.01$ and $r = 0.9$. We consider two cases for pollution:

- All the filters are polluted (dashed curve).
- Only the last i filters are polluted. This number varies from 1 to 9 (dotted curves).

We observe from Figure 5.8 that, if only the last filter is polluted, *i.e.*, the first 9 filters remain untouched, then the blue curve and the dashed curve superimpose for most of the part. Clearly, as more and more filters are polluted, the eventual false positive probability achieved by the attacker is increased.

[6] www.phishtank.com

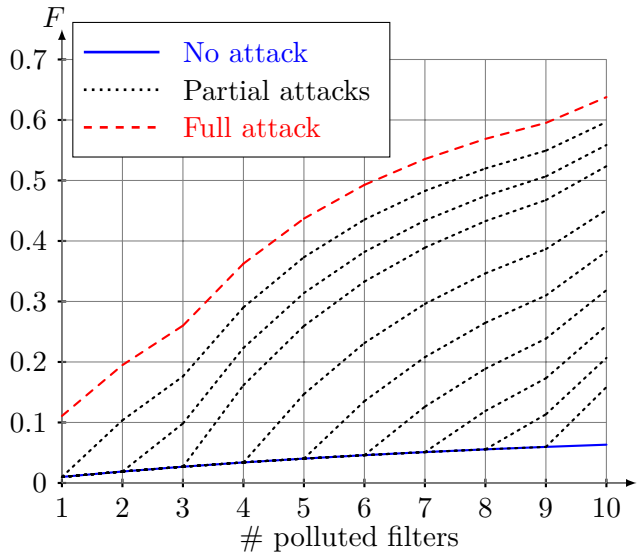

Figure 5.8: Polluting Dablooms. F represents the combined false positive probability (see Chapter 2). The last curve in black from left represents the false positive probability achieved when only the last filter is polluted.

Deletion attacks on Dablooms are also feasible due to the presence of Counting Bloom filters. The forgery of the required URLs is straightforward since MurmurHash can be inverted in constant time (see [AB12]). Furthermore, by exploiting counter overflows, an adversary can force Dablooms to create an *empty filter*, even after all the n items have been inserted. Let us define $a = nk \bmod 16$ and b such that $nk = a + 16b$. The goal of the adversary is to obtain after n insertions a filter set to 0 everywhere except one counter set to the value a. All the other insertions make overflow b other counters. The insertion counter of the filter may say it is full while none of the values inserted will be detected in the future. Such a filter is a complete waste of memory. Empty filters make Dablooms bigger and useless.

5.4 Networking Tools

There are several network-related tools that implement Bloom filters for tasks such as caching, resource routing, packet inspection, *etc.* In this section, we discuss two such tools namely, Squid which is a web proxy and AIEngine which is a network intrusion detection system.

5.4.1 Web Proxy: Squid

Web proxies are means to efficiently handle the daunting task of managing web traffic and alleviating network bottleneck. These proxies reduce the bandwidth consumption by relying on caching and reusing frequently requested web pages. Achieving the full benefits of the proxy mechanism requires them to co-operate and share their caches. Squid [Wes04] is a notable example of a caching proxy leveraging *cache digests* [RW98], a summary of the contents of the *Internet Object Caching Server*. A Squid server first stores the fetched

pages in a hash table and then at regular intervals of 1 hour, transforms it into a space-efficient cache digest which could later be shared with the peers. When a request for a URL is received from a client, a cache can use digests from its peers to determine if any of them have previously fetched the URL. The cache then requests the object from the closest peer.

Crosby and Wallach [CW03] have previously attacked the hash table in version 2.5 STABLE-1. Our work extends their attack to cache digests in version 3.4.6.

Cache digests in Squid are built as a Bloom filter. Although the protocol design allows for a variable number of hash functions, in practice, Squid employs 4 hash functions for the "sake of efficiency" and dissuades developers from using more. Furthermore, instead of computing 4 independent hash functions over a URL, Squid generates a 128-bit MD5 hash of the key (comprising of the URL and the HTTP retrieval method) and then splits it to obtain the indexes of the filter. For unexplained reasons, the filter parameters are not optimal. In fact, to insert n items in the filter, the considered filter size is $5n + 7$, instead of the optimal size $6n$. This choice leads to a higher false positive probability. For instance, if $n = 200$, the entailed false positive probability is 0.09 instead of the optimal value 0.03, hence an increase by a factor of 3.

Attacking Squid's cache digest requires us to set up a client, two Squid proxies as siblings in a LAN and an HTTP server responding to every GET request of the client received via one of these proxies. When the proxies are configured as siblings, they work together and exchange cache digests to avoid unnecessary hits between them. Each unnecessary hit between proxies costs bandwidth and adds latency to the client's request. We suppose that the proxies are honest, this avoids the trivial attack where a proxy transmits a fake filter to its peers. Our attack relies on a malicious client who generates fake URLs and asks one of the proxies to fetch these pages. These URLs pollute the cache digest of the concerned proxy. To pollute the cache digest of one of these proxies, we used an HTTP server outside our LAN which responds to every HTTP GET request, and then forges URLs at our convenience. Once the cache digest of the first proxy is fully built, we start querying the second proxy and count the cache digest false positives (*i.e.*, unnecessary hits to the first proxy). Each false positive adds at least one round-trip time (10 ms in our setup) between the two caches to the response delay. With 100 URLs added to pollute a clean cache digest (51 URLs are already present when the cache is totally clean), the filter size is 762 bits. We observed that out of 100 queries, cache pollution increases the false positives hits to 79% in contrast to 40% when the cache is unpolluted.

5.4.2 Intrusion Detection System: AIEngine

Artificial Intelligent Engine *aka* AIEngine[7] is an open-source packet inspection engine with capabilities of learning without any human intervention. It serves as a general purpose NIDS (Network Intrusion Detection System) or as a network forensic tool. In its restricted form, it may also provide the functionality of a DNS domain classification or a network collector among several others.

[7] https://bitbucket.com/camp0/aiengine

AIEngine includes an implementation of Bloom filters borrowed from the `boost`[8] library. The `boost` C++ library by default uses an implementation of the hash function object specified by the Draft Technical Report on C++ Library Extensions (TR1) [Boo05], and employs seeding to simulate multiple hash functions. Bloom filters in AIEngine are referred to as `IPBloomSets`. As the name suggests, the purpose of an `IPBloomSet` is to store a whitelist/blacklist of IP addresses, which can later be exploited by a network administrator to detect and act upon unauthorized activities. For instance, AIEngine comes with an example code that can be used *as-it-is* to detect Tor activity on the network by storing a list of IP addresses in an `IPBloomSet`.

Let us assume that a network administrator maintains a whitelist of email addresses used to filter spam emails. The addresses are stored in an `IPBloomSet` such that when an email message is received, the sender's address is checked against the filter. If the result is negative, it is marked as a spam. In this case, the result of the queries are public, as an adversary might check whether his emails are marked as spam. For example, the adversary can spam his personal email account and see if the messages are being filtered. This renders a false positive forgery attack easy to mount. After a sequence of queries to the filter, the adversary might be able to find a bulk of email addresses that are not marked as spam although they are not in the whitelist, and thus bypass the security of the AIEngine and flood users with spam emails.

While no recommended parameters have been included with the AIEngine source code, the `boost` library provides several example codes that build filters with parameters: $m = 8192$ bits and $n = 5000$, which gives a false positive probability of 0.45. For these parameters, an adversary needs no significant effort to forge a false positive since every second randomly chosen item should serve the purpose with high probability.

5.5 Forensic Tools

Bloom filters are ideal to store large blacklists (or whitelists) of malicious (non-malicious) identifiers, making them a popular choice for forensic tools, where the essential task is to detect "known-goods" on a target's hard disk. In the following, we study two such tools: NSRL filter and `sdhash`.

5.5.1 NSRL Forensic Filter

The National Software Reference Library (NSRL) [Nat16] is based out of the National Institute of Standards and Technology (NIST). It is designed to collect software from various sources and incorporate file profiles computed from the software into a Reference Data Set (RDS) of information. The RDS can then be used by law enforcement, government, and industry organizations to review files on a computer by matching file profiles in the RDS. The goal is to alleviate the effort during triage of files and hard drives that have been seized as a part of criminal investigations.

[8] https://github.com/cabrera/boost-bloom-filters/

Farrell *et al.* [JGW08] propose to compute SHA-1 digests of files in RDS and store them in a Bloom filter that can be distributed to forensic experts. A file on a subject's disk can then be checked against the filter to determine whether the file is a "known good".

An adversary in the context of forensic analysis may wish to mount a query-only attack with the goal of either forging a false positive or generating a dummy query. The latter may for instance allow a criminal to buy time for further damage. Even worse, the ability to forge a false positive allows an attacker to completely bypass the forensic analysis and hide a contraband file. Placing ourselves in a related context, let us imagine an adversary owning a contraband file that she wishes to hide from the forensic tool. In order to achieve this, she may easily and reversibly modify the file such that it yields a false positive when queried to the filter.

Considering the large size of RDS 2.19, which is around 14 million, the authors suggest to use a Bloom filter with $m = 2^{32}$, $k = 5$ and SHA-1 hash function [JGW08]. SHA-1 produces a 160-bits digest which is then divided into 5 smaller chunks each of 32 bits that simulate 5 independent hash functions. The filter has a false positive probability of 8.08×10^{-10}. The C, C++ and Java source code of NSRL Bloom filter is available online.[9] It is to be noted that the C and C++ code provide the possibility to use an HMAC instead of a hash function, but in the default setting HMAC is not used. Moreover, the Java code does not provide any such functionality.

The current RDS version is RDS 2.51 (updated on 12-01-2015), however since the filter parameters have not been updated, we work with RDS 2.19 as done in [JGW08]. The attacks clearly hold even for the newest version of the dataset. We consider a `.gz` file as a contraband target. The choice of a `.gz` file is motivated by two factors. First, that any file can be compressed to obtain a `.gz` file. Second, that the `GZIP` file format reserves 4 bytes to store the uncompressed input size modulo 2^{32}. Since, the false positive probability of the filter is $8.08 \times 10^{-10} > 2^{-32}$, out of 2^{32} randomly chosen items, at least one item yields a false positive with very high probability. In other words, the 4 bytes provide enough number of bits to forge a false positive. Moreover, changing these bytes only affects the meta data and does not change the actual content of the file. We employ a naive approach to forge a false positive by exhaustively searching for the value of these bytes. For each set of 4 bytes, we compute the SHA-1 digest of the resultant `.gz` file and query the filter for its belonging. Our search took around 3 hours using the C implementation without HMAC. It is pertinent to note that an HMAC provides no additional security against our false positive forgery attacks since the success probability of the attack also depends on the false positive probability of the filter.

5.5.2 Extension to sdhash

Limitations of the NSRL filtering mechanism becomes apparent, when one attempts to find an embedded object inside a document or an archive — file-level hashes are useless. In order to overcome these limitations, Roussev [Rou10] proposes `sdhash`. It is a full fledged forensic tool that allows two arbitrary blobs of data to be compared for similarity

[9] https://github.com/simsong/frag_find

based on common strings of binary data. `sdhash` is currently used in the very popular Sleuthkit[10] and Viper[11].

The tool first identifies *statistically-improbable features* of a "file of interest", which are then inserted into a Bloom filter. A feature is a binary data block, usually of 64 bytes. The filter uses SHA-1 hash function with $k = 5$. The SHA-1 digest of a feature is split into 5 sub-hashes, which are then used as independent hashes during insertions and queries. The implementation uses several 256-byte filters with $n = 128$ items per filter. Thus, the expected false positive rate of the filters is 0.0014. In fact, the effective number of filters used depends on the number of features to be represented. After a filter reaches its capacity, a new filter is created and the process is repeated until the entire set of features is represented. Finally, to compare two files `sdhash` compares their respective Bloom filters. In general, the overlap between two compatible filters (same size and built using same parameters) is a measure of the overlap between the sets they represent [BM05].

We reiterate that from a set of items used to construct a Bloom filter, it is possible to find another one which constructs the same filter. This attack consists in creating useless files by finding many groups of false positives sharing the same Bloom filter than a file of interest. Thus, the identification of a file of interest is rendered impossible with the tool. Given a Bloom filter corresponding to a file, we first generate 128 blocks that are false positives as in query-only attacks. These blocks are then plugged into a file such that the blocks are considered as statistically-improbable features. In practice, we populate the file to ensure that all the recently found blocks have the lowest normalized Shannon entropy — a metric that is used in `sdhash` to determine statistically-improbable features.

5.6 Predicting Unknown Filter Parameters

In our adversarial models, we assumed that the filter parameters are public as we argued that it holds for open-source implementations. Furthermore, the adversary was given access to the source code of the application, so security through obscurity was not acceptable.

In the following, we discuss the scenario of proprietary software solutions, where the filter parameters are not necessarily available to the adversary. The issue is equally relevant in situations where the filter is hosted on the server side and the parameters are not made public. We explore whether the filter parameters can be learned by an adversary when only given an oracle access to the filter. In the following, we focus on the problem of determining k, the hash functions and m; since once these are known, n follows using (2.2). We further note that the knowledge of filter parameters suffices to learn the internal state of the filter.

5.6.1 Number of Hash functions

Let us assume that an adversary wishes to learn the number of hash functions, *i.e.*, the parameter k of the Bloom filter. We further assume that the adversary only has an oracle access to the filter, *i.e.*, she may query the filter as many times as she wants by presenting

[10] http://www.sleuthkit.org/

[11] https://digital-forensics.sans.org/blog/2014/06/04/managing-and-exploring-malware-samples-with-viper

any item of her choice. The oracle in return answers with 1 if the item is present in the filter and 0 otherwise. It is assumed that no filter parameter is known to the adversary. We note that these assumptions hold even in case of proprietary software.

In order to estimate the value of k, the adversary performs the following straightforward experiment. She picks a randomly chosen item from $\{0,1\}^*$, and queries the oracle for the item. The probability that the oracle answers positively for a randomly chosen item should be the same as the false positive probability of the filter. Repeating the experiment with sufficiently large number of items should yield a close estimate of the false positive probability. Once, the false positive probability f_{opt} is ascertained, the value of k can be immediately determined using the relation $k_{\text{opt}} = -\log_2(f_{\text{opt}})$. The results of our experiments are shown in Table 5.2. As argued, the adversary can indeed estimate k with high accuracy.

Table 5.2: Results of our experiments for filters of capacity 1 million and with varying number of hash functions, k. We present the expected and the observed number of false positives for 10 million random items. k_{est} shows the value estimated by the adversary using the relation: $k_{\text{est}} = -\log_2(\text{Observed \#false positives}/10^7)$.

k	Expected #false positives	Observed #false positives	k_{est}
2	2500000.0	2496165	2.002
3	1250000.0	1249873	3.000
4	625000.0	625835	3.998
5	312500.0	312438	5.000
6	156250.0	157613	5.987
7	78125.0	78059	7.001
8	39062.5	39260	7.992
9	19531.25	19342	9.014
10	9765.625	9914	9.978

5.6.2 Filter Size

Determining the size of the filter is in general a very difficult task if not impossible. However, if an adversary knows the number of items that have been inserted into the filter without knowing the actual capacity of the filter, she can determine m. In order to do this, the adversary performs the same experiment as the one described to determine k. This allows her to learn f and k. Now, since she also knows the number of items that have been inserted into the filter, she may determine m using (2.1).

In the general case, the adversary has three options: 1) guess the value of m; 2) mount attacks by presenting inputs/queries that work for several different values of m; and 3) mount attacks by sending streams of attack data, where each stream is designed for one particular m.

Finding inputs/queries that work for multiple values of m is not practical; the search space grows proportionally to the least common multiple of the candidate filter sizes. This may easily exceed the space of digests of the underlying hash functions, rendering the attacks infeasible. However, if the number of candidate filter sizes is small enough, then the attacker can compute separate attack streams aimed at each m. For a chosen-

insertion attack, this clearly creates some limits to the scope of the attack. In fact, due to the uniformity of hash functions, during the initial phase of the attack, any stream should maximize the number of set bits. But, as more and more items are inserted into the filter, multiple streams can quickly exhaust the capacity of the filter. Separate attack streams can however be useful for finding a dummy query. This however incurs a linear (in the number of streams) computational overhead.

5.6.3 Hash Functions

An attacker may also obtain some information about the hash function used in a Bloom filter by relying on the so called "remote timing attacks". Remote timing attacks [BB03, BT11] have been successfully employed in the past to extract private keys from a network-based OpenSSL web server. In a slightly different setting, Tobin and Malone [TM12] employ remote timing attacks to identify unknown hash functions used in a hash table.

In case of Bloom filters, the idea is to time the query to a filter and compare it with the computation of a hash function. In fact, the attacker first benchmarks a set of hash functions and compares them with the time needed by an oracle to process a query. Since, the most costly operation in the query processing is the cost of computing digests, this in general allows the attacker to learn some information about the hash function.

In Figure 5.9, we time the query to sample Bloom filters built using different hash functions. The experiments were done on a 64-bit laptop computer powered by an Intel i7 3520M processor at 2.90GHz with 4MB cache and running 3.8.0-35 Ubuntu Linux. We found that the cost of computing the digest of the items were very close to these timings, a difference of roughly 0.3μs. We first note that a filter built using MD5/SHA-1 is roughly two times faster than the one with SHA-256 and roughly 3 times faster than the one with SHA-384/SHA-512. However, we observe that MD5 and SHA-1 are hard to distinguish. The same holds for SHA-512 and SHA-384. In situations where the ambiguity in identifying the hash function cannot be resolved, an attacker may generate streams of attack data, where each stream is designed for a particular hash function. While the computational overhead in this case may not be prohibiting, it could nevertheless reduce the impact of a chosen-insertion attack. In fact, during the initial phase of the attack, any stream should maximize the number of set bits. However, if the ambiguity is due to 2 hash functions, half of the later insertions may turn to be wasteful.

In the previous discussion, we implicitly assumed that k different hash functions are used. In situations where only a couple of hash functions are employed to simulate the remaining, the adversary must compare the query time with the time required to make the effective number of calls to the hash function. For instance, in case of `sdhash`, only one hash function simulates 5 hash functions. This implies that the adversary has to compare the time required to compute one digest with the time to query the filter. However, learning which one of the (several) hashing simulations have been employed is quite difficult. This is because for instance, two calls to MD5 require roughly the same time as one call to SHA-256 that simulates two hash functions.

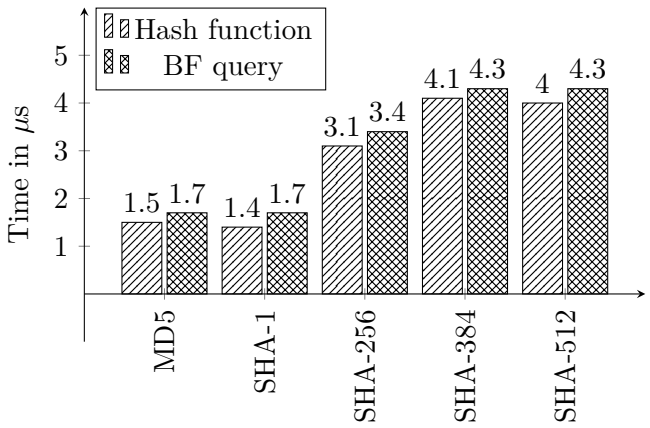

Figure 5.9: Comparing the cost of evaluating a hash function and the cost of querying a Bloom filter, when built using the same hash function. Filter parameters are $k = 10, n = 10^6$, and item size is 32 bytes.

5.7 Countermeasures

We explore different solutions to render Bloom filter enabled applications resistant to adversaries. A first and obvious solution would be to employ an alternate data structure. Intuitively, this may come with a potential risk of losing the vital benefit of Bloom filters namely, low memory footprint. To this end, we study another data structure proposed by Bloom in the original paper on Bloom filters [Blo70]. It turns out that this data structure performs better than Bloom filters in resisting attacks while incurring no overhead. An alternative solution would be to keep using Bloom filters but with better parameters, which can be recomputed from (5.2). It gives us the worst-case parameters for Bloom filters. Choosing these parameters increases the memory consumption, but does not prevent developers from using non-cryptographic hash functions. It further defeats the chosen-input adversary but not the query-only one. Another solution consists in using keyed-hash functions. It defeats all classes of adversaries but increases the query time. In the following, we discuss these countermeasures in further detail.

5.7.1 Alternate Data Structures

Since, all our attacks rely on the fact that Bloom filters are a probabilistic data structure, the most immediate countermeasure to our attacks would be to use an alternate data structure such as *hardened hash tables* [CW03] that use universal hash functions. Such data structures are immune to all kinds of algorithmic complexity attacks. However, unlike Bloom filters, hash tables entail a higher memory overhead as they require storing items *per se*. This hence renders hash tables less attractive compared to Bloom filters.

In this section, we study a replacement data structure which is a hybrid of Bloom filters and hash tables. It is structurally similar to hash tables but much like Bloom filters, it is probabilistic and entails false positives. The data structure is originally due to Bloom [Blo70], and hence we refer to it as *Bloom Hash Tables*. Surprisingly, despite their

appealing properties, Bloom hash tables have not received much attention. In fact, they appear to have been largely eclipsed by their sister data structure, *i.e.*, Bloom filters.[12] Our results reaffirm that not only this data structure is often more memory efficient than Bloom filters, but they also resist better to our attacks.

5.7.1.1 Description

In the following, we briefly describe Bloom hash tables. For a more detailed treatment, interested readers may refer to [Blo70, CFG+78]. In the following, we let $k > 0$, to be an integer such that Bloom hash tables have a false positive probability of 2^{-k}. We abusively use k here despite the fact that the variable k was used to denote the number of hash functions in a Bloom filter. The reason being that, extending the same notation here allows us to denote the false positive probability of Bloom hash tables to be 2^{-k}, which is in accordance with the error probability of Bloom filters with optimal parameters.

As in the case of Bloom filters, let $\mathcal{S} = \{x_1, \ldots, x_n\}$ be a set of n items to be represented by the table. Hence, n defines the capacity of the Bloom hash table. A Bloom hash table is in fact a hash table with $a \times n$ buckets, where,

$$a = 1 + \frac{1}{k + \log_2(k)}.$$

The items in $\mathcal{S}$ are the *keys* of the hash table, while the *values* are small digests of $k+\log_2(k)$ bits.

Let h be an $(k+\log_2(k))$-bit hash function that excludes the value 0, and let $h_1, h_2, \ldots$ be an independent infinite sequence of hash functions mapping $\{0,1\}^*$ into bucket indices $\{0, 1, \ldots, (a \times n) - 1\}$. The first hash function hashes the item to generate the value to be stored in the bucket, while the second sequence of hash functions is used to determine the bucket index where the value is stored.

Insertion. The table is initially empty, *i.e.*, all the buckets contain the value 0. Each of the n items x_i is inserted into the hash table by finding an empty bucket. For the first item x_1, the value $h(x_1)$ is inserted at the index $h_1(x_1)$; for the second item x_2, the value $h(x_2)$ is stored in the first bucket $h_1(x_2), h_2(x_2), h_3(x_2), \ldots$, for which the value stored is 0. More generally, an empty bucket is found by checking indices $h_1(x_i), h_2(x_i), h_3(x_i), \ldots$ until an index j is found for which the value at the index $h_j(x_i)$ is 0.

Query. In order to test for an item y, one needs to compute $h(y)$ and compare with the values at indices $h_1(y), h_2(y), h_3(y), \ldots$ until an index j is found such that the value stored at $h_j(y)$ equals 0 or $h(y)$. In the former case, the item is not present in the hash table, while in the latter, y is assumed to be present with a small false positive probability 2^{-k}.

Figure 5.10 schematically represents a Bloom hash table.

[12] The only plausible reason to explain this could be the simplicity of Bloom filters compared to a slightly more involved Bloom hash tables.

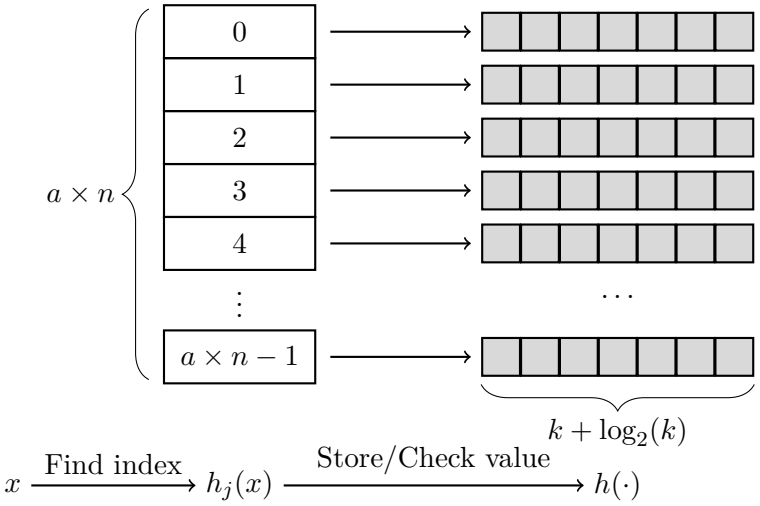

Figure 5.10: Bloom hash table data structure.

5.7.1.2 Bloom Filters versus Bloom Hash Tables

We know that for optimal parameters, a Bloom filter with a false positive probability of 2^{-k} that stores n items requires $m = 1.44nk$ bits of space. While for the same parameters, a Bloom hash table requires a total storage of:

$$m' = n \times a \times (k + \log_2(k)) = n \times (k + \log_2(k) + 1)\text{bits},$$

which is basically the size of the index array times the size of the digest generated by h. We plot the ratio of m and m' in Figure 5.11. For $k > 10$, Bloom hash tables are more economical than Bloom filters. For smaller values of k, the two data structures are comparable.

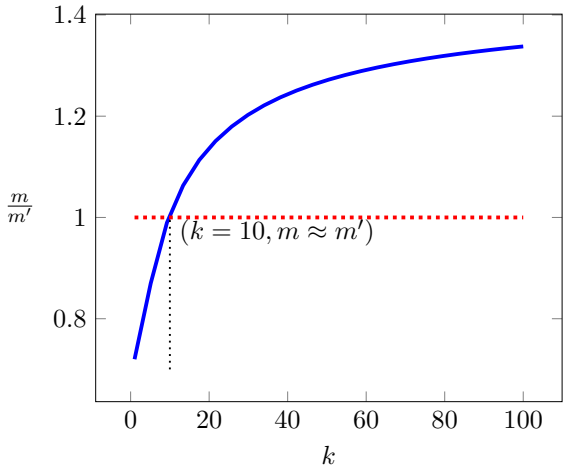

Figure 5.11: Memory comparison for Bloom filters (m) and Bloom hash tables (m').

As for the computational cost, Bloom filters requires $\mathcal{O}(k)$ computations of hash functions in the worst case (when an item is in the filter). While, Bloom hash tables require $\mathcal{O}(k)$ hash computations in average for items not in the table, while in average $\mathcal{O}(\ln(k))$ for items in the table (see [CFG+78] for details).

This comparison clearly suggests that Bloom hash tables can prove to be a better data structure compared to Bloom filters for smaller false positive probabilities.

5.7.1.3 Attacks against Bloom Hash Tables

In the following, we first explore chosen insertion attacks against Bloom hash tables and then in the sequel we briefly discuss query-only attacks.

Recall that classical Bloom filters with optimal parameters can not be saturated, *i.e.*, rendered in a state where every item from the universe is considered to be in the filter. Bloom hash tables on the other hand can be saturated even with the recommended parameters. In order for Bloom hash tables to be saturated, each possible value must be stored in the table and none of them should be 0. Since each value is a digest on $(k + \log_2(k))$ bits. There are $2^{k+\log_2(k)} - 2$ possible values in total (excluding 0). In order for these values to be present in the table, the table must itself be of appropriate size. In fact, in order to store all the values in the table, one must have:

$$\begin{aligned} a \times n &\geq 2^{k+\log_2(k)} - 2, \\ \Rightarrow n &\geq \frac{2^{k+\log_2(k)} - 2}{1 + \frac{1}{k+\log_2(k)}}. \end{aligned} \tag{5.5}$$

Hence, as soon as the number of items inserted in the table verifies the above inequality, the table will become fully saturated. For instance, when $k = 8$, which gives a false positive probability of 2^{-8}, the minimum number of items required to completely saturate the table is 1876.

However, unlike Bloom filters, Bloom hash tables resist better to saturation attacks. In order to see this, we compare the minimum number of polluting items required to saturate either of the data structures initialized with identical parameters. Let us consider an example where we fix the parameters as follows: $n = 500$ and $k = 10$. While the choice of n is arbitrary here, k has been chosen such that $m \approx m'$. With this choice of parameters both the data structures contain the same number of items and entail the same false positive probability. A simple computation shows that the corresponding Bloom hash table has a size of 7161 bits, while the corresponding optimal Bloom filter will have 7213 bits. Using these parameters and (5.3), (5.5), the minimum number of polluting items required to saturate the Bloom hash table is 9524, while only 721 items suffice for a Bloom filter. Hence, the Bloom hash table requires 13 times more items than required by the Bloom filter with identical parameters. This example highlights that while both the data structures require comparable space, Bloom hash tables resist better to pollution attacks.

More generally, if n' counts the minimum number of polluting items for a Bloom hash table, then the same number also corresponds for a Bloom filter if:

$$n' \times k \geq m = \frac{nk}{\ln 2},$$

which means the filter is optimal for a capacity of $n' \ln 2$.

As for query-only only attacks which require finding false positives, both the data structures provide the same resistance. However, in order to fully defend against such attacks the false positive probability must be small enough to guarantee cryptographic security — 2^{-80} being the recommended value. For a false positive probability of this order, Bloom hash tables can prove to be more economical in terms of space savings than Bloom filters as shown in Figure 5.11.

5.7.2 Worst-case Parameters for Bloom Filters Or *How to minimize an adversary's advantage?*

While designing an application based on Bloom filters, the developer has two constraints: m, the memory available and n, the capacity of the filter. Based on these parameters, he minimizes the false positive probability and obtains the optimal number of hash functions to employ (2.2). The adversary on the other hand tries to increase this false positive probability by inserting specially crafted items (5.2).

An adaptive approach to the pollution attacks could be to choose the parameters such that the adversary's advantage could be minimized. In other words, the developer would fix m and n as earlier, but now instead of finding k that minimizes the false positive probability of the filter, he chooses k that minimizes the false positive probability envisaged by the adversary, *i.e.*, $f^{\text{ADV}} = \left(\frac{nk}{m}\right)^k$. Differentiating f^{ADV} with respect to k gives:

$$\frac{\partial f}{\partial k}^{\text{ADV}} = f^{\text{ADV}} \cdot \left(1 + \ln\left(\frac{nk}{m}\right)\right) \quad .$$

The zero of the derivative is:

$$k_{\text{opt}}^{\text{ADV}} = \frac{m}{en} \quad , \tag{5.6}$$

and the second derivative test confirms that $k_{\text{opt}}^{\text{ADV}}$ is the point of minimum of f^{ADV}. The false positive probability achieved by the adversary would then be:

$$f_{\text{opt}}^{\text{ADV}} = e^{-m/en} \quad . \tag{5.7}$$

However, given n and m, and assuming that the optimal value $k = k_{\text{opt}}^{\text{ADV}}$ is used by the developer, the actual false positive probability of the filter in this worst case scenario is obtained by using this k in (2.1):

$$f_{\text{worst}} = \left(1 - e^{-1/e}\right)^{\frac{m}{ne}} \quad , \tag{5.8}$$

$$\ln\left(f_{\text{worst}}\right) = -0.433\frac{m}{n} \quad . \tag{5.9}$$

From (2.2), (2.3) and the obtained results, we highlight that:

$$\frac{k_{\text{opt}}}{k_{\text{opt}}^{\text{ADV}}} = e \ln 2 = 1.88 \quad \text{and} \quad \frac{f_{\text{worst}}}{f_{\text{opt}}} = (1.05)^{\frac{m}{n}} \quad .$$

Consequently, the adaptive approach requires a considerably lesser number of hash functions and hence, the associated Bloom filter would be more time efficient at the cost

of a slightly higher false positive probability.

Let us now consider a scenario where the developer for a given m and n chooses $k_{\text{opt}}^{\text{ADV}}$ to obtain the false positive probability of the filter satisfying (5.9). However, if the developer wishes to obtain the same false positive probability, while only fixing n and not the size of the filter, the new filter size $\bar{m}$ can be obtained from (2.3). Comparing these two filter sizes for the same false positive probability gives: $\frac{\bar{m}}{m} = 4.8$.

Hence, we observe that the filter size increases almost by a factor of 5 in the latter case. With this solution, developers can keep their fast non-cryptographic hash functions but at the cost of a larger Bloom filter. It defeats chosen-insertion adversaries but not the query-only ones.

5.7.3 Probabilistic Solutions

The first countermeasure proposed to defeat algorithmic complexity attacks was to use *universal hash functions* [CW77]. These were empirically studied in case of hash tables. We propose to use a *Hash-based Message Authentication Code* (HMAC) [MVO96], which has been considered an overkill until now (see [CW03] and the references therein).

We assume that the server running the Bloom filter works with an HMAC. The server chooses a key for the HMAC at the beginning and uses it to insert/check items submitted by clients. The key is chosen from a large set, therefore it is computationally infeasible for an adversary to either guess the key or compute pre-images for all the possible values. Thus, the adversaries defined in Section 5.2 can not make a brute force search to satisfy conditions (5.1) or conditions (5.4) because the function is no longer predictable. Hence, attacks against Scrapy, Dabloom and Squid can easily be mitigated since in all these applications, the Bloom filter is stored on the server side. We however note that if given an oracle access to the filter, the attacker may still mount false positive forgery attacks. The reason being that the success probability of the attack also depends on the parameter f of the filter. Consequently, in order to defend against these attacks, f must be chosen to render a brute force search infeasible. We recommend $f \leq 2^{-80}$.

The crucial question that remains is the impact of probabilistic solutions on the processing time of a Bloom filter. We focus on the benchmark of cryptographic hash functions and HMAC for Bloom filters. It is commonly admitted that those functions are too slow compared to non-cryptographic hash functions and hence are rarely employed. We decided to make a fair comparison between the two classes of functions. We observed that in several implementations of Bloom filters, there are k calls to the hash function with different salts. This in fact forces many bits of the digests to remain unused. One could reuse these bits to reduce the number of calls to the expensive hash function and obtain comparative results. As all cryptographic hash functions pass the NIST test suite to check their uniformity, we can assume that all the bits of the digest can be used. The indexes of an item require having $k \cdot (\lfloor \log_2 m \rfloor + 1)$ bits of digests in total. Figure 5.12 shows the domain of application of the different cryptographic hash functions (and thus their respective HMAC construction) for different false positive probabilities. A single call to SHA-512 or HMAC-SHA-512 is enough to build any Bloom filter with optimal parameters

for $f \geq 2^{-15}$ and m smaller than 1GB. For $f \leq 2^{-20}$, we need to make several calls to the hash function.

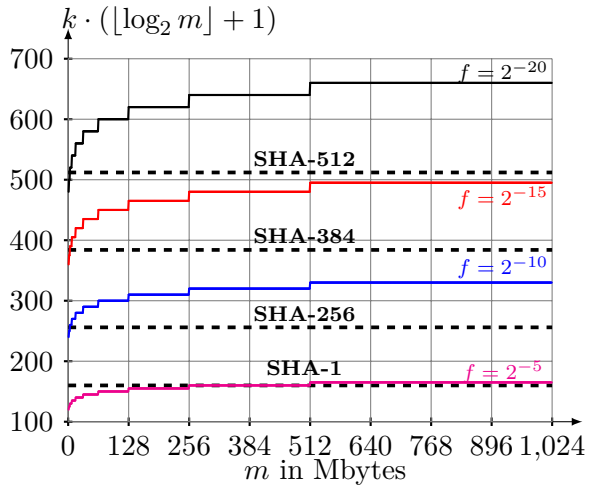

Figure 5.12: Domain of application of hash functions.

Table 5.3 presents a comparative summary of the cost of reusing bits of different cryptographic hash functions and HMAC-SHA-1. Our implementations are in C and the cryptographic primitives are taken from the OpenSSL (version 1.0.1) library. The filter has a false positive probability of 2^{-10} and contains 1 million elements, creating a filter of size 2.48MB. The items inserted in the filter are of 32 bytes long (corresponding to SHA-256 prefixes). Clearly, recycling of bits performs significantly better than the naive $k = 10$ calls to the hash function. HMAC-SHA-1 is costlier due to the 2 implicit calls to SHA-1. Compared to the popular choice of MurmurHash, recycling is a better and more secure alternative. SipHash [AB12], a non-cryptographic keyed hash function outperforms HMAC-SHA-1 by a factor 7 without recycling. Our technique reduces the gap to a factor 4 making HMAC affordable.

Table 5.3: Time to query a filter.

	Timing (μs)		
Hash function	**Naive**	**Recycling**	**Speedup** ($\times$)
MurmurHash-32 [App10]	0.7	-	-
MD5	5.9	0.28	21
SHA-1	6	0.29	20.6
SHA-256	51	0.49	104
SHA-384	53.3	0.78	68
SHA-512	53.6	0.8	67
HMAC-SHA-1	11.8	1.2	9.83
SipHash [AB12]	1.7	0.3	5.66

HMACs have the advantage to defeat all the adversaries and to keep the original parameters of Bloom filters. The drawback is that the query time is increased compare to non-cryptographic hash functions.

5.8 Related Work

Below we discuss studies related to the security of Bloom filters from two angles: attacks and defense.

5.8.1 Attacks

The most pertinent literature on the security of Bloom filters is a recent work by Naor and Yogev [NY15]. The authors analyze the false positive probability in an adversarial model, where the adversary can *adaptively* query the filter, *i.e.*, she can submit queries depending on previous responses. The fundamental goal of the adversary is to forge a false positive. The authors present a tight connection between Bloom filters in this model and *one-way functions* in cryptography. Despite having similar goals, our work distinguishes itself from [NY15] in two aspects: 1) While [NY15] provides a purely theoretical framework to study Bloom filters under adversarial settings, our work measures the impact on real-world applications. 2) The authors consider only a query-only adversarial model, we on the other hand, consider several adversaries with different capabilities. Furthermore, the adversarial model considered in [NY15] is weak in the sense that, an adversary only has an oracle access to the filter and cannot see its internal memory representation. In contrast with [NY15], our adversarial models are stronger and assumes that the adversary can even insert items into the filter and has access to its internal state.

Our attacks on `sdhash` is similar to the one discussed in [JGW08]. The paper provides a performance evaluation of Bloom filters in the context of forensic analysis and profiling of hard drives. The analysis relies on a Bloom filter that stores SHA-1 hashes of files considered to be "known and safe". The authors consider the scenario where an attacker wishes to hide a contraband JPEG file. To this end, she modifies the file such that it yields a false positive when queried to the filter. The authors propose to use an HMAC to defend against such attacks, but admit that it reduces the usability of the tool. We argue that the power of the adversary has been underestimated in [JGW08]. In fact, the attack reappears when the adversarial model is strengthened, *i.e.*, by giving an oracle access to the filter. The success of the attack is due to the high false positive probability (in cryptographic terms), which apparently cannot be defended by employing an HMAC. We further extend this work by defining several other adversary models.

In [BBB12] Breitinger *et al.* describe an attack against `sdhash`, called Bloom filter shifts, this is an *anti-blacklisting* attack, in the sense that it consists in modifying an incriminated file in a manner that the score of the comparison between the original signature and the signature produced with the modified file is approximatively 25%. Due to a low score, the file can eventually be declared as safe. We extend the scope of this attack by proposing a new pollution attack. We create many files which share the same signatures. The goal is to flood the tool with useless files, so the incriminated file is surrounded by many useless files. The filtering process fails because too many files remain: the time spent to identify an important file is greatly increased.

Our work retains the flavor of *algorithmic complexity attacks*, which were first intro-

duced in [Pes98] and formally described by Crosby and Wallach in [CW03]. The goal of algorithmic complexity attacks is to force an algorithm to run in the worst case execution time instead of running in average time. For a hash table, it means that the search operation runs in linear time instead in constant time. The general impact of these attacks was DoS [MR04](due to the significant consumption of resources) or the creation of covert channels [SCZ11].

Algorithmic complexity attacks have successfully been mounted against many data structures and algorithms: hash tables [CW03, BYW07, KW11, BPBBLP12, AB12], quicksort [McI99], skip-lists [BR09], machine learning [HJN+11], regular expressions [Cro03], packet analyzers [PPM12] and file-systems [CGJ09]. In most of these works, weak non-cryptographic hash functions were responsible for the attacks. Our work contributes to algorithmic complexity attacks in three aspects. First, it extends algorithmic complexity attacks to a data structure having false positives. Second, our pollution attacks and false positive flooding are simpler and relatively much easier to mount compared to the pre-image attacks described by Crosby and Wallach [CW03]. They choose a bucket identifier, id in the hash table and search for keys $x_1, \ldots, x_n$ which go to this bucket, *i.e.*, $h(x_1) = \cdots = h(x_n) = id$ (multiple pre-images). Third, we provide different adversary models to analyze Bloom filters. These models encompass previous attacks and hence can be re-used to analyze other data structures. For instance, the attacks on hash table and skip-lists combine a chosen-insertion and query-only adversary. For attacks against quick-sort and regular expressions, it is a chosen-insertion adversary.

The problem of saturation of Bloom filters is well-identified in the software development community, typically in the situations where the number of insertions is not controlled. We show the best strategy for an adversary to pollute and saturate a filter in the chosen-insertion model and present the entailed complexity. Similarly, the use of Bloom filters is often criticized by the web crawler community due to the intrinsic false-positives (see [ON10]). Our work materializes those criticisms and shows how fast they may arise.

5.8.2 Defense

Several countermeasures have been proposed to prevent algorithmic complexity attacks. Crosby and Wallach [CW03] suggest using universal hash functions [CW77]. They are also used in the Bloom filter included in the Heritrix web spider [Her16]. Aumasson *et al.* have applied several tools from cryptanalysis in [AB12] to attack non-cryptographic hash functions. They also propose a new function SipHash [AB12] as an efficient and secure alternative. It is interesting to notice that Venkataraman *et al.* [VSGB05] design probabilistic counting algorithms for fast detection of superspreaders, and pay attention to the implementation: *"We use the OPENSSL implementation of the SHA-1 hash function, picking a random key during each run, so that the adversary cannot predict the hashing values."* However, they also advert the usage of non-cryptographic hash functions: *"For a real implementation, one can use a more efficient hash function."* The authors in [GXT+08] have also used keyed hash functions to compute packet statistics in the presence of an adversary.

We also note that, our chosen-insertion attacks should trivially extend to CM sketches. To this end, we explored several existing implementations, but all of those use the universal hash functions proposed by Carter and Wegman [CW77].[13] Because of the use of universal hash functions, the attacker cannot predict the hash functions that will be used for an item and hence cannot succeed in mounting attacks.

An abounding literature is devoted on designing secure Bloom filters [NK09, Ker11] or private Bloom filters [BC04]. The underlying approach is to replace the usual hash functions by group ciphers. Bellovin *et al.* [BC04] use Pohlig-Hellman encryption for instance. These cryptographic primitives resist pre-image and second pre-image attacks at the cost of a higher computational time. Kerschbaum [Ker11] has used partially homomorphic encryption to make private queries to a Bloom filter. Unfortunately, the scheme is computationally intensive, and hence can not be adapted to build high performance data structures.

Särelä *et al.* [SRZ+10] study the security of multicast protocols based on Bloom filters. In these protocols, the Bloom filter represents the group entities. The authors propose a technique called *BloomCasting*, which enables controlled multicast packet forwarding. In order to control the entities who may send/receive packets to/from a group, the authors suggest to use keyed-hash functions (among other possible alternatives such as a secret permutation) for Bloom filters.

Our idea of recycling bits of cryptographic digests is inspired by the Nyberg's accumulator [Nyb96], an alternative to Bloom filters. Nyberg employs a *"long hash function"* that produces digests of arbitrary size. All the bits of the digest are used to build an accumulator having a size comparable to that of a Bloom filter. We keep the idea of a *"long hash function"* and implement it by salting a cryptographic hash function and by recycling bits. By ensuring that all the bits of the hash functions are used, one can prevent against our pre-image and second pre-images based attacks. Our attacks against Bloom filters do not hold against Nyberg's accumulator because it would require finding pre-images for full digests of cryptographic hash functions. However, Nyberg's accumulator requires a hash function that generates digests of megabytes in size (for recommended parameters) which makes it less attractive to developers. To obtain long digests, Nyberg suggests to combine cryptographic hash functions and pseudo-random generators. We salt and recycle bits.

The recommendations of the NIST [Dan12] on the usage of cryptographic hash functions is a reference document for truncated digests. However, developers are still ignoring the threats of short truncated digests. RFC 6920 [FKD+13] also dedicates a few paragraphs to the threats associated with truncated digests, but it gives no specific consequences. Our work gives a concrete example of these threats.

[13] We believe that this is probably because the authors of the CM sketch proposed the first implementation which employed universal hash functions. Their code was open-source and hence eventually influenced all future implementations.

5.9 Summary

In this work, we present several attacks against Bloom filters. Our results show that Bloom filters should be judiciously deployed as these seemingly "innocuous" data structures can be easily exploited and can be forced to behave as per the terms of an adversary. Our findings demonstrate that if Bloom filter parameters are ill-chosen, they are prone to severe algorithmic complexity attacks leading to DoS. We also provide new filter parameters to be used to minimize the effect of attacks and discuss the entailed trade-off. We also observe that Bloom hash tables — a hybrid data structure that combines Bloom filters and hash tables can prove to be a promising data structure to replace Bloom filters.

Part II

Security and Privacy of Safe Browsing

Chapter 6

Safe Browsing

Contents

6.1 Online Phishing and Malware Threats **103**

6.2 Overview of Safe Browsing Services **105**

6.3 Google Safe Browsing **108**

6.3.1 Lookup API 110

6.3.2 Safe Browsing API v3 112

6.3.3 Local Data Structures 115

6.3.4 Safe Browsing Cookie 117

6.3.5 Lookup API versus Safe Browsing API 118

6.4 Facilitating Safe Browsing Usage **118**

6.4.1 Safe Browsing Diagnostic Tool 119

6.4.2 Reporting Tools 119

6.5 Effectiveness and Usability Studies **121**

6.6 Yandex Safe Browsing **121**

6.7 Summary **122**

6.1 Online Phishing and Malware Threats

As the web plays an ever increasing role in information exchange, so too is it becoming the prevailing platform to mount attacks. As a result, online users are often the targets of *phishing* and *malware* attacks. In a phishing attack, the goal is to steal money or personal and sensitive information such as passwords and credit card details. Malware (short for malicious software) on the other hand has a much wider goal. Apart from harvesting information, stealing money and identities as in phishing attacks, malware attacks are particularly characterized by their attempt to trick online users into installing unwanted and malicious software. The ultimate goal is to deny users access to essential electronic resources, disrupt the functioning of large information systems or surreptitiously modify the integrity of a stored data.

The general modus operandi of the attackers is the following: masquerade as a trustworthy entity and present seemingly authentic URLs or web pages to unsuspecting users. These URLs host phishing or malware content and are circulated among potential victims through online channels such as emails. To increase the success rate, some attacks combine phishing with malware for a blended attack. For instance, a potential victim may receive a phishing e-card via email that appears to be legitimate. By clicking on the link inside the email to receive the card, the person is redirected to a spoofed website which downloads an unwanted software to the victim's computer.

Web-based phishing and malware attacks have become popular among attackers since it has become much easier today to setup and deploy websites, and inject malicious code through JavaScript and iFrames [PMRM08]. In fact, even unskilled attackers can rely on the off-the-shelf phishing and malware kits provided by a thriving cyber crime ecosystem. It further appears that cyber criminals are migrating to a new business model known as Malware-as-a-Service (MaaS), where authors of exploit kits offer extra services to customers in addition to the kit itself. The future is bleak as preliminary results from Symantec (published in 2008) suggested that "*the release of malicious code and other unwanted programs may be exceeding that of legitimate software applications*".

Due to their wide reach, phishing and malware attacks have formed the basis of a very lucrative business. Indeed, over the last 20 years, malware has evolved from occasional "exploits" to a global multi-million dollar criminal industry. As for phishing, RSA Security LLC estimated in its October 2011 Fraud Report that the worldwide losses from phishing attacks cost more than $520 million in the first half of 2011 alone.

Phishing and malware threats affect all actors alike, be it governments, businesses or individuals. As governments rely ever more on the Internet to provide services for citizens, they face complex challenges in securing information systems and networks from attack or penetration by malicious actors. Governments are also being called on by the public to intervene and protect consumers from online threats such as ID theft. Businesses have been equally affected as a study presented in the Kaspersky Security Bulletin 2015 concludes that 29% of computers — *i.e.*, almost every third business-owned computer was subjected to one or more web-based attacks.

While there is no silver bullet, several technologies at different levels have been employed to combat phishing and malware threats. This includes spreading social awareness to avoid being a victim, employing technology to detect such threats in real time and defining stricter legislations to punish the perpetrators. Training people to recognize phishing attempts and malware threats has largely relied on visually discerning fake and authentic web pages. In face with educated users, attackers have become even more motivated and hence construct dangerously convincing URLs and web pages to remain inconspicuous over the Internet. To this end, technologies such as Secure Socket Layer (SSL) and Extended Validation (EV) SSL help to fight phishing and other online frauds. SSL protects data in motion, which can otherwise be intercepted and tampered with if sent unencrypted. Extended Validation (EV) SSL complements security by providing a reliable way to establish trust via certificates. The certificate issued by a trusted party provides a tangible proof to

online users that the site is indeed legitimate and is safe to transact with. Similar solutions to detect websites hosting malware have also been developed. These solutions perform a security scan of the given website and detect malicious behavior. The scan may require running the software in a sandboxed environment. Once the web page is diagnosed to be hosting malware, it may then be included into a blacklist for a faster decision making in the future.

This chapter presents a specific class of anti-phishing and anti-malware technology for web-based threats, called *Safe Browsing* (SB). While the name Safe Browsing mainly refers to the technology conceived by Google, we abusively use the term to denote similar technologies developed by other companies. In the following, we first provide an overview of SB services provided by companies other than Google. In the sequel, we provide a comprehensive description of Google Safe Browsing and Yandex Safe Browsing — two of the most popular SB services. Details provided here form the basis of the following chapters where security and privacy aspects of these SB services are analyzed.

6.2 Overview of Safe Browsing Services

All SB services are implemented at the application layer (HTTP level) of the Internet stack. Additionally, SB services share a common design principle: filter malicious links by relying either on a blacklist of malicious URLs, domains and IP addresses or a whitelist of *safe-to-navigate* websites. In fact, whenever, a user visits a web page or queries a search engine, the browser requests the service provider to lookup in the blacklist stored on the server. The server then sends a response depending on which users can be warned about the potential danger of visiting a malicious page. Since SB features are often included in the browsers, the service has been implemented in a way that it does not affect the browser's usability. The robustness of the service often relies on fast lookup data structures which may generate false positives.

It is to note that the blacklists provided by the SB servers are extremely dynamic. This is due to the fluctuating behavior of malicious domains: a safe-to-navigate domain transforming into a malicious one and vice versa. A safe-to-navigate domain turns malicious when attackers inject malicious code into it, while a malicious domain becomes safe when harmful codes get cleaned up.

In the following, we briefly present some popular SB services which include Microsoft *SmartScreen URL Filter* [Mic11a], *Web of Trust* [WOT15b], Norton *Safe Web* [Sym15b] and McAfee *SiteAdvisor* [McA15a]. The SB services presented in this section either recover each URL visited by a user, or a part of it (such as the domain name). These services hence are not privacy friendly by design.

Microsoft SmartScreen Filter. SmartScreen filter [Mic11a] is a reputation-based filter included in Internet Explorer 8 (IE8). It checks URLs against a whitelist, and requires the client to send the URL to the server and obtain a response. It also checks downloaded files: the SHA-256 digest of the file is sent to the server along with other information. Since the release of Windows 8, SmartScreen has become a system feature of Windows,

and hence is not solely restricted to IE. With the advent of IE9, a new feature has been added to the SmartScreen: *SmartScreen Application Reputation*. It aims to protect users from unknown malware programs. This software reputation scheme is backboned by the *Authenticode signatures* [Mic11b, Mic16].

MICROSOFT's privacy policy for IE reads: "*When you use SmartScreen Filter, the address of the website you are visiting will be sent to* MICROSOFT *with standard computer information. Search terms or data that you entered in forms, might be included.*" For files, MICROSOFT has a similar statement. From time to time, information about the usage of SmartScreen Filter is also sent to MICROSOFT, such as the time and total number of websites browsed since an address was sent to MICROSOFT for analysis. "*Some information about files that you downloaded from the Web, such as name and file path, may also be sent to Microsoft. Some website addresses that are sent to Microsoft may be stored along with additional information, including web browser version, operating system version, SmartScreen Filter version, the browser language ...*". MICROSOFT asserts that the aforementioned information are only used to analyze performance and to improve the service.

Web of Trust. Web of Trust (WOT) [WOT15b] is a free browser add-on. The add-on makes query to a reputation database to know whether or not a website is trustworthy. It uses several sources including Phishtank[1], GOOGLE and YANDEX Safe Browsing (explained later in the chapter). The browser add-on can be used to learn about the status of a URL (see Figure 6.1), or to annotate the results of a query to the preferred search engine (see Figure 6.2).

The add-on automatically creates a random identifier for a user. At each query, this identifier, the date and time of the request and the hostname of each site a user visits are sent to the WOT servers to recover the rating information.

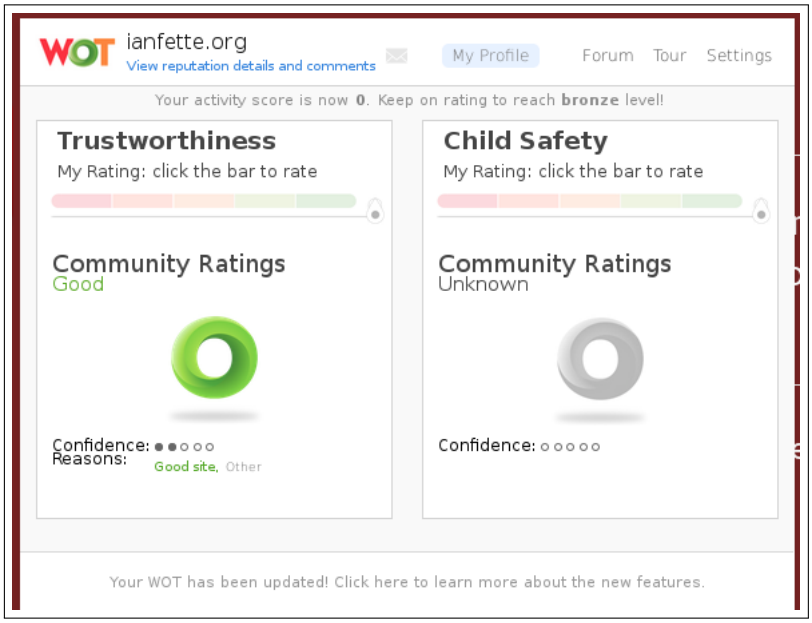

Figure 6.1: Status of ianfette.org on WOT.

[1] phishtank.com

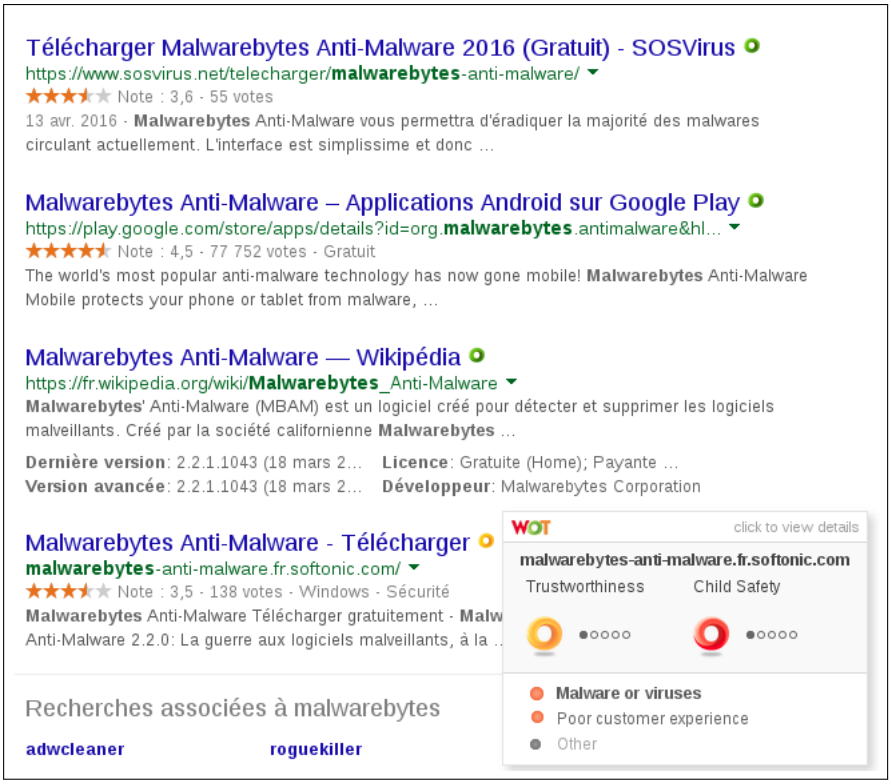

Figure 6.2: WOT annotates results of a search query for the keyword 'malware bytes'.

Norton Safe Web. Norton Safe Web (NSW) [Sym15b] is developed by Symantec and delivers information about websites based on automated analysis and user feedback. Safe Web operates as a browser plugin and requires IE6 or Firefox 3 or later or Chrome. Recently, Symantec has teamed up with *Ask.com* to provide on top of the Safe Web, a search environment called Norton Safe Search (NSS) [Sym15a] which offers visual site ratings within search results (see Figure 6.3, Figure 6.4). Even when NSS is disabled, NSW continues to annotate searches on the default search engine.

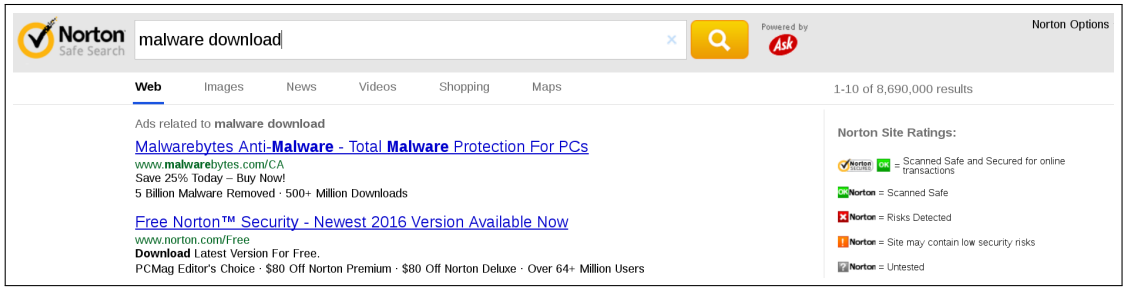

Figure 6.3: Norton Safe Search for Chrome.

In addition to actively crawling and analyzing websites, NSW relies on feedback from a network of more than 20 million Norton Community Watch end-points that automatically submit in real-time suspicious URLs to the NSW server for detailed analysis. When a drive-by-download occurs at a website, the suspicious URL is automatically reported to Norton Safe Web for analysis. The reported site is rated as unsafe if the analysis confirms that the download is malicious. In order to ensure that its site rating accurately reflects

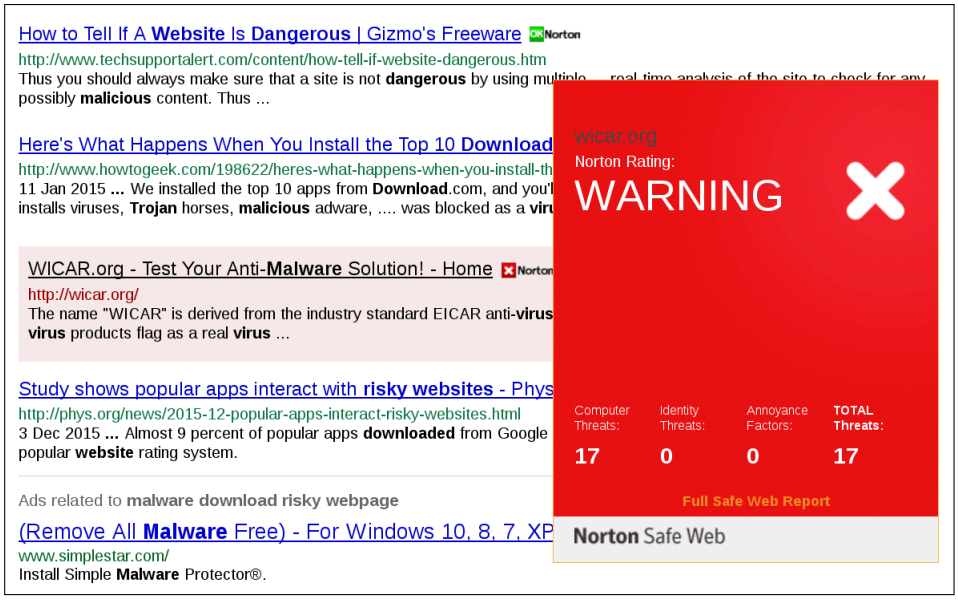

Figure 6.4: Norton Safe Search annotates results of a search query for the keyword 'malware download'.

the current state of a site, NSW employs an "*intelligent aging*" process to look at a site's history to determine how frequently it should be re-analyzed. Symantec asserts that: "*Ask.com collects information including: IP address, the origin of the search …and may share this information with a third party*".

McAfee SiteAdvisor. McAfee SiteAdvisor (SA) [McA15a] is a plugin available for Windows (Firefox, Internet Explorer and Chrome) and for Mac OS (Firefox and Safari). It provides a secure search box and gives one of these 4 ratings to each site, *Safe*, *Caution*, *Warning*, *Unknown*. An extended and paid version of the software comes with SA Live [McA15c]. With the SA Live software, even emails and instant messages get checked for malicious links. It can also scan downloaded files for threats. Depending on the chosen protection level, download protection may stop the download and warn users of the risks. Protection against malicious downloads is not available in the standard SA freeware. SA employs a lookup mechanism to contact the rating server [McA15b] and generates a log file which is sent to the ePolicy Orchestrator for reporting purposes. SA currently sends logs only through HTTP which can be easily eavesdropped.

The following section presents a comprehensive description of Google Safe Browsing. Our special focus on the service by Google stems from the fact that its service is the most popular of all and unlike afore-presented SB services, Google Safe Browsing has been built with the goal of reducing the privacy leakage.

6.3 Google Safe Browsing

Google Safe Browsing (GSB) is the most popular SB service. It was developed by Google in 2008. The GSB architecture has been carefully designed to be able to inform users in real time. As a consequence, GSB has proved to be a very effective and popular tool

to combat web-based phishing and malware threats. According to a 2012 report [Pro12], Google detects over 9,500 new malicious websites everyday and provides warnings for about 300 thousand malware downloads per day. Furthermore, the service has gathered an extremely large user base: Google claims more than a billion users of its SB services until date [Goo14c]. The large user base is mainly due to the integration of SB as a security feature in all major browsers such as Google Chrome, Mozilla Firefox, Safari and Opera. According to Statcounter[2], these represent 65% of all the browsers in use. It is pertinent to note that GSB is not restricted to search, but it also extends to ads. It has further opened paths for new services such as instantaneous download protection, *i.e.*, protection against malicious *drive-by-downloads*, chrome extension for malware scanning and Android application protection.

The essential goal of GSB is to warn and dissuade end users from visiting malicious URLs. In fact, whenever a client (typically a browser) attempts to visit a malicious URL, the client can display an interstitial warning page before the suspicious web page is actually requested. Figure 6.5 shows the warning page displayed in Mozilla Firefox for `ianfette.org`. The page also offers the possibility to ignore the warning and proceed to the requested web page. This is particularly useful in situations where the web page gets incorrectly categorized as malicious.

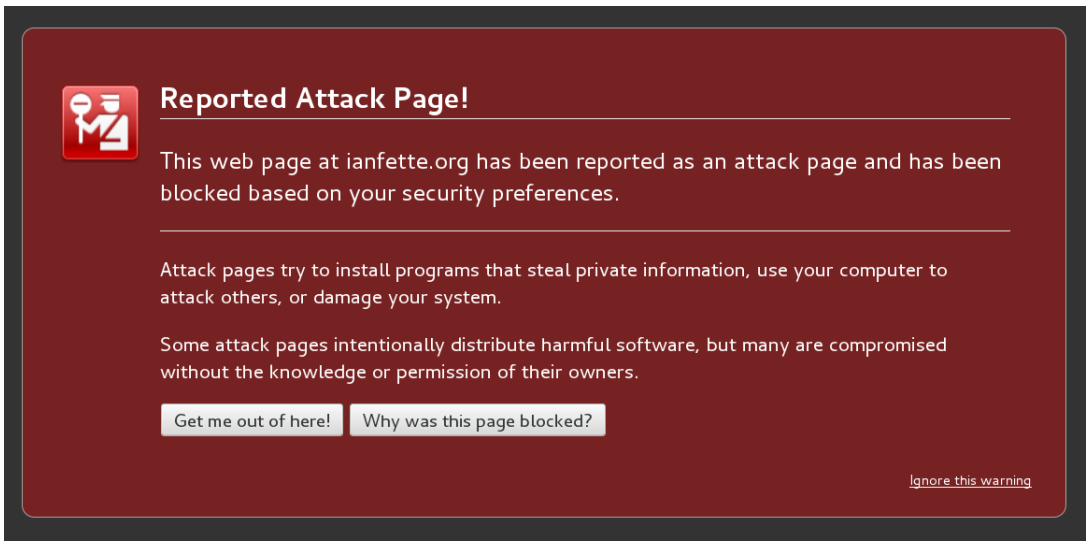

Figure 6.5: Warning page displayed to the user for a web page hosting malware.

Figure 6.6 shows a simplified architecture of the service. Google crawlers continuously harvest malicious URLs on the web and then transmit them to the GSB server, which maintains a blacklist of malicious web pages. Clients can then consult the server to check if a link is malicious.

Google classifies malicious web pages into three main categories: malware, phishing and unwanted software, and hence it maintains multiple blacklists. All the blacklists and the number of entries per list are given in Table 6.1. It is to be noted that the lists are extremely dynamic, which means that the number of entries change very frequently.

[2] `statcounter.com`

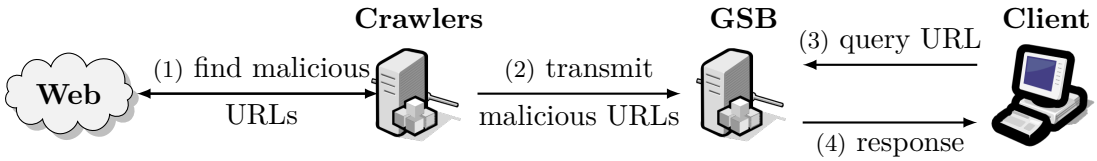

Figure 6.6: High level overview of Google Safe Browsing.

In fact, the lists contain SHA-256 [Nat12] digests of malicious web pages, which include complete URLs, domains and IP addresses. The lists can be accessed by clients using two different APIs. The choice of the right API is left to software developers who choose the one they prefer according to the constraints they have. The following two sections describe in detail these APIs.

Table 6.1: Lists provided by Google Safe Browsing. Information could not be obtained for cells marked with *.

List name	Description	#Entries
goog-malware-shavar	malware	317,807
goog-regtest-shavar	test file	29,667
goog-unwanted-shavar	unwanted software	*
goog-whitedomain-shavar	unused	1
googpub-phish-shavar	phishing	312,621

6.3.1 Lookup API

The *Lookup* API [Goo15b] is a simple interface to query the state of a URL. Clients send URLs they need to check using HTTP GET or POST requests and the server's response contains a direct answer for each URL. The response is generated by looking up in the malicious lists maintained on the server. In the following, the two query methods are described in detail.

6.3.1.1 GET Method

A sample `GET` request to the GSB server looks like: `https://sb-ssl.google.com/safebrowsing/api/lookup?client=demo-app&key=12345&appver=1.5.2&pver=3.1&url=ianfette.org`. The protocol requires the following CGI parameters in the `GET` request:

1. The `'client'` parameter indicates the name of the client that invokes the request.
2. Each client registers on the Google Developers Console[3] to obtain an API key. The API key identifies the client and is used to enforce quota and control access to the service. The `'key'` parameter in the request specifies this API key.
3. The `'appver'` parameter indicates the version of the client.
4. The `'pver'` parameter indicates the protocol version supported by the client. This should be '3.1'.

[3] https://console.developers.google.com/

5. The 'url' parameter indicates the URL to lookup. The URL must be valid as per RFC 1738 [BLMM94]. The URL queried for in the sample request is ianfette.org.

Since, ianfette.org is currently listed as a domain hosting malware, the SB server sends back the following response with the response body containing the keyword 'malware'.

```
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Content-Length: 7
------------------------------------
malware
```

6.3.1.2 POST Method

A POST request is identical to the GET request except that the 'url' parameter is not provided. A sample POST request is of the form: https://sb-ssl.google.com/safebrowsing/api/lookup?client=demo-app&key=12345&appver=1.5.2&pver=3.1. The POST method specifies the queried URLs in the request body using the following format:

```
POST_REQ_BODY = NUM LF URL (LF URL)*
NUM = (DIGIT)+
URL = URL string
```

The request body contains several lines separated by a Line Feed (LF). The first line is a number (NUM) that indicates how many URLs are included in the body. This number must match with the number of URLs listed. The lines below are the URLs to be looked up. There must be one URL per line, and at least one URL overall (empty lines do not count). The URLs must be valid, but need not be encoded. A client can query up to 500 URLs in a single POST request. For instance, a request body for 2 URLs looks like:

```
2
google.com
ianfette.org
```

For a POST request, the server returns the URL type for each URL in the response body, if at least one of the queried URLs is found in the suspected phishing, malware, or unwanted software lists:

```
POST_RESP_BODY = VERDICT (LF VERDICT)*
VERDICT = "phishing" | "malware" | "unwanted" | "phishing,malware" |
"phishing,unwanted" | "malware,unwanted" | "phishing,malware,unwanted" | "ok"
```

For our example of a request body with 2 URLs, the response body is:

```
ok
malware
```

Access to the GSB service using the Lookup API is controlled to prevent DoS attacks or other potentially abusive uses. This has been implemented by enforcing a quota per API key. In fact, a single API key can make requests for up to 10,000 clients per 24-hour period. The number of different clients that can be supported with a single API key is limited to 10,000 per day.

6.3.2 Safe Browsing API v3

The GOOGLE Safe Browsing API v3 [Goo15a] is another API to use GSB. It is to be noted that the Safe Browsing API v2 is deprecated, and v1 has been discontinued. We note that all the work described in this dissertation focuses on Safe Browsing API v3. However, we also note that GOOGLE has released Safe Browsing API v4 in June, 2016 (post publication of our work). In general, the architecture of v4 is very similar to that of v3 and includes a few new features and modifications. Though not formally acknowledged by GOOGLE, some of the work described in this dissertation on v3 have influenced the changes in v4. We defer the discussion on the changes made in v4 to Chapter 8. And, unless otherwise stated, API v3 is assumed in the rest of this dissertation.

The Safe Browsing API is much more sophisticated than the Lookup API. In contrast to the Lookup API, the client now does not handle a URL directly. URLs are in fact processed through several steps. The foremost being *canonicalization*, which is the procedure to sanitize a given URL. To this end, the URL is first validated according to the URI specifications [BLFM05]. If the URL uses an internationalized domain name (IDN), it should be converted to the ASCII *Punycode* representation. The URL must also include a path component; that is, it must have a trailing slash (`http://example.com/`). Tab (`0x09`), carriage return (`0x0d`) and line feed (`0x0a`) characters are removed from the URL. Resource fragments at the end of the URL are also removed. The same is done for any percent-escapes, and all characters that are $\leq$ ASCII 32, $\geq$ 127, "`#`", or "`%`" are percent-escaped. The escapes should use uppercase hex characters.

Once the URL is sanitized, the hostname and the path in a URL are canonicalized independently. The following steps are followed to canonicalize a hostname:

1. All leading and trailing dots are removed.
2. Consecutive dots are replaced with a single dot.
3. Any uppercase character is replaced by its lowercase.
4. If the hostname can be parsed as an IP address, it is normalized to 4 dot-separated decimal values.
5. Username and passwords prepended with the hostname are removed.

While to canonicalize the path, the client proceeds in the following manner:

1. The sequences "`/../`" and "`/./`" in the path should be resolved by replacing "`/./`" with "`/`", and removing "`/../`" along with the preceding path component.

2. Runs of consecutive slashes are replaced with a single slash character.

3. Query parameters are left untouched.

The second step consists in generating *decompositions* of the canonicalized URL. A decomposition is a URL composed of subdomains and subpaths of the target URL. For the sake of illustration, let us consider the most generic HTTP URL (before canonicalization) of the form `http://usr:pwd@a.b.c:port/1/2.ext?param=1#frags` as defined in [BLMM94], where `usr` is a username and `pwd` is the corresponding password, `a.b.c` is a fully-qualified domain name, `port` is a TCP or UDP port number, `1/2.ext` is the URL path, `param=1` is the query and `#frags` identifies a specific place within a remote resource. All the possible decompositions of the generic URL in the order they are generated are given below. Notice that the resource identifier, the username and the password have been removed during the canonicalization step.

```
1 a.b.c/1/2.ext?param=1
2 a.b.c/1/2.ext
3 a.b.c/1/
4 a.b.c/
5 b.c/1/2.ext?param=1
6 b.c/1/2.ext
7 b.c/1/
8 b.c/
```

It is in the third step, that the actual decision on the status of a URL is taken. In this step, the client computes a SHA-256 digest for each decomposition. Each digest is then checked against a locally stored database which contains 32-bit prefixes of malicious URL digests. If the 32-bit prefix of the digest is not found to be present in the local database, then the URL can be considered safe. However, if there is a match, the queried URL may not necessarily be malicious: it can be a false positive. This is because distinct SHA-256 digests may share a common 32-bit prefix due to hash collisions. Consequently, the client needs to eliminate the ambiguity and hence queries the GSB server by sending the prefix. The server in response sends all the full digests corresponding to the received prefix. Finally, if the full digest of the client's prefix is not present in the list returned by the server, the URL can be considered safe. Figure 6.7 summarizes a request through the GSB API.

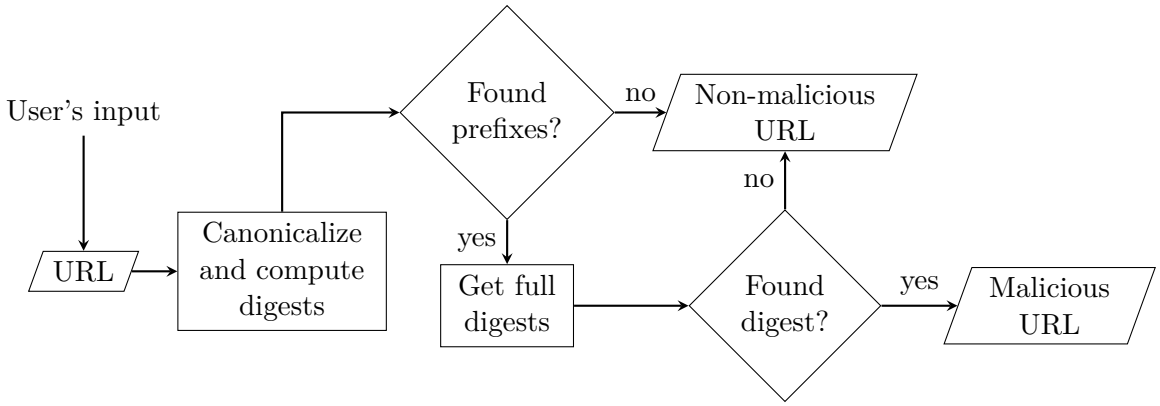

Figure 6.7: Google Safe Browsing API: Client's behavior flow chart.

The client performs a lookup for decompositions in the given order. The lookup for

all the decompositions is required since the complete URL might not have been included in the blacklists.

A decomposition is said to create a *hit* if the corresponding prefix is present in the local database, otherwise it is said to create a *miss*. If any of the decompositions creates a hit in the local database then the initial link is considered as suspicious and the prefix can be forwarded to the Google server for a confirmation. If there are more than 1 hits, then all the corresponding prefixes are sent. After receiving the list of full digests corresponding to the suspected prefixes fragments, they are locally stored until an update discards them or when the client restarts, whichever is the earliest. Storing the full digests prevents the network from slowing down due to frequent requests. To maintain the quality of service and limit the amount of resources needed to run the API, Google has defined for each type of request (malware, phishing, or unwanted software) the frequency of queries that clients must restrain to (see [Goo15a] for more details).

It is to note that at the beginning the local database of the client does not exist and has to be initialized and updated. This may take a while until the database is fully synced and during this time, it is possible for a client to miss a malicious URL because it does not yet know the corresponding prefix. As of now, the API gives the first 32 bits of SHA-256 digests to be locally stored, but a client must be able to handle any length up to 256 bits.

Data format. Prefix blacklists stored on the client's side are comprised of a series of *chunks*, the smallest unit of data that is sent to the client. This allows for supporting partial updates to all users, including new users, and allows for more flexibility in choosing which data to send. The actual chunk size is determined by the server. Chunks are of two kinds: "add chunk" and "sub chunk". An "add chunk" contains new entries for the list, while a "sub chunk" contains entries that need to be removed from the client's list. A chunk is identified by an ID number, which is a sequence number for chunks of the same type. The total number of "add" and "sub" chunks are generally different. The server does not explicitly list all hashes that need to be removed. Instead, to save bandwidth, the server indicates which chunks need to be deleted by specifying a previously-seen "add" or "sub" chunk ID number. When a client updates his database, he sends the lists of the chunks' ID numbers that he already owns and gets back only the new chunks. In the same response, he retrieves the minimum delay he has to wait before the next update request.

The following shows the request format, where each part is recursively defined.

```
BODY      = [SIZE LF] (LIST LF)+ EOF
SIZE      = "s;" DIGIT+                 # Optional size, in kilobytes and >= 1
LIST      = LISTNAME ";" LISTINFO (":" LISTINFO)*
LISTINFO  = CHUNKTYPE ":" CHUNKLIST
CHUNKTYPE = "a" | "s"                   # 'Add' or 'Sub' chunks
CHUNKLIST = (RANGE | NUMBER) ["," CHUNKLIST]
NUMBER    = DIGIT+                      # Chunk number >= 1
RANGE     = NUMBER "-" NUMBER
```

And the following presents an example of a request body, where the client requests a response size of 200 kilobytes for the two given lists. It then lists the chunks it already

has for each list type.

```
s;200
googpub-phish-shavar;a:1-3,5,8:s:4-5
goog-malware-shavar;a:1-7:s:1-2
```

In response, the server sends a wait time before the next request can be sent. Among other information, the response contains redirect URLs from where the data can be retrieved.

6.3.3 Local Data Structures

In order to ensure the efficiency and scalability of the service, an efficient data structure to store the prefixes is a necessity. The choice of the right data structure is often constrained by two factors: fast query time and low memory footprint. While fast query time ensures the usability and throughput of the service, memory usage is critical for web browsers in memory constrained mobile environments. Google has tested two different data structures until now: *Bloom filters* [Blo70] and *Delta-coded tables* [MKD+02].

6.3.3.1 Bloom Filters in Chromium

In an earlier version of Chromium (discontinued since September 2012), a Bloom filter was used to store the prefixes' database on the client's side. However, the false positive probability in this case was an issue because it could force clients to perform requests more often than necessary to get full digests for a prefix, eventually leading to a heavier load on Google servers. To minimize this, the Chromium development team used a Bloom filter with 20 hash functions. The hash function used is the one proposed by Jenkins [Jen96]. They also set the minimum size of the Bloom filter to 25,000 Bytes and the maximum size to 3 MB. The filter can then accommodate between 8,000 and 1,006,632 items (on an average 25 bits per item). The false positive probability expected in this range can be computed using standard filter equations, which is approximately 6.57×10^{-6}. We note that these parameters are not optimal, as for a filter size varying from 25,000 Bytes to 3 MB, and the number of hash functions being 20, the optimal number of elements that can be added should lie between 6,931 and 872,180 (*Cf.* Chapter 2 § 2.4).

6.3.3.2 Delta-Coded Tables

The Chromium development team switched to another data structure in the later releases. The new data structure is called *delta-coded tables* [MKD+02]. Unlike classical Bloom filters, this data structure is dynamic, does not have any "intrinsic" false positives, and yet incurs a lower memory footprint. However, its query time is slower than that of Bloom filters. The balance between the query time and the memory usage seems suitable within the operational constraints. Even though delta-coded tables do not entail false positives, its use to store 32-bit prefixes in Chromium indeed leads to false positives. False positives arise due to the fact that several URLs may share the same 32-bit prefix.

In case of delta-coded prefix tables, digest prefixes are sorted, and the difference of the consecutive prefixes is computed. The final representation requires two arrays, storing prefix differences using 16 bits. An index handles the case when 16 bits can not encode a difference and provides a quicker random access. A maximum of 100 differences between each index is allowed. For further detail on the data structure, interested readers may refer to RFC 3229 [MKD+02].

Example 6.3.1 (Delta-Coded Prefix Tables) *Figure 6.8 presents a schematic representation of this example. In this example, let us consider the following set of five sorted prefixes:* $\{20, 100, 60000, 200000, 210000\}$. *Two vectors are considered: index vector and delta vector which are initially empty (see Step 1 of Figure 6.8). To store the prefixes into the delta-coded table, first the difference set is computed* i.e., $diff = \{20, 80, 59900, 140000, 10000\}$. *A pair composed of the first prefix and the size of the initial delta vector* i.e., $(20, 0)$ *is then inserted into the index vector (Step 2). Since, the subset* $\{80, 59900\}$ *of the prefixes can be encoded using less than 16 bits as both 80 and 59900 are less that* $2^{16} - 1$, *the delta vector now contains the list of these differences* i.e., $\{80, 59900\}$ *(Step 3). The following difference, 140000 however cannot be encoded using 16 bits, as it is greater than* $2^{16} - 1$, *hence a new index is assigned, and the pair composed of the prefix and the current size of the delta vector* $(200000, 2)$ *is inserted in the index vector (Step 4). Finally, the last delta 10000 is inserted in the delta vector since it can be encoded using 16 bits (Step 5).*

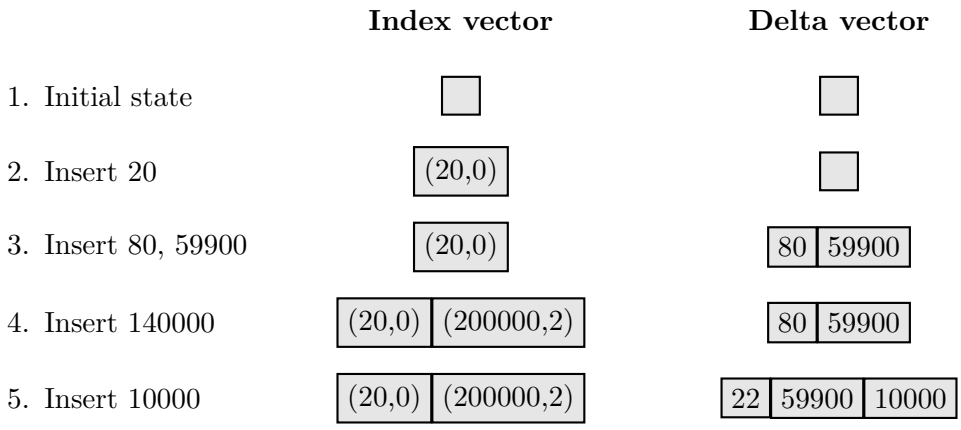

Figure 6.8: Evolution of the index vector and the delta vector in a delta-coded prefix table. At step (1), the vectors are empty. The final state of the vectors is shown in step (5).

We have implemented both the data structures to understand why Bloom filters were abandoned and the rationale behind the choice of 32-bit prefixes. The results are shown in Table 6.2. If 32-bit prefixes are stored, the raw data requires 2.5 MB of space. Storing these prefixes using a delta-coded table only requires 1.3 MB of memory, hence Google achieves a compression ratio of 1.9. Bloom filters are immune to the change in the prefix size, and thus require a larger yet constant space of 3 MB. However, starting from 64-bit prefixes, Bloom filters outperform delta-coded tables. The low memory footprint of Bloom filters comes at the cost of being a static data structure, which makes them unsuitable for

the highly dynamic blacklists provided by Google. Indeed, a prefix cannot be removed from a Bloom filter however new ones can be added. Delta-coded tables clearly permit both inclusion and removal of prefixes. This may require recomputing certain differences and updating entries of the index and delta vectors. The property of being dynamic and memory efficient partly justifies Google's choice of delta-coded tables over Bloom filters and the choice of 32-bit prefixes. It is to note that the choice of 32-bit prefixes is however also influenced by the network bandwidth consumed by SB clients and servers.

Table 6.2: Client's local database size for different prefix sizes.

		Data structure (MB)			
		Delta-coded table		Bloom filter	
Prefix size (bits)	**Raw data (MB)**	**size**	**comp. ratio**	**size**	**comp. ratio**
32	2.5	1.3	1.9		0.8
64	5.1	3.9	1.3		1.7
80	6.4	5.1	1.2	3	2.1
128	10.2	8.9	1.1		3.4
256	20.3	19.1	1.06		6.7

6.3.4 Safe Browsing Cookie

GSB stores a mandatory preference cookie on a client's computer. When implemented inside web browsers, each request to the API also sends this cookie. The cookie sent by the browsers is the same as the one used by other services provided by Google especially the social features such as the *+1 button*. In the following, we present the contents of the cookie for Firefox. The cookie actually consists of two SQL tables: `meta` and `cookies`.

```
TABLE meta(key LONGVARCHAR NOT NULL UNIQUE PRIMARY KEY, value LONGVARCHAR);

TABLE cookies(creation_utc INTEGER NOT NULL UNIQUE PRIMARY KEY,
                host_key TEXT NOT NULL, name TEXT NOT NULL,
                value TEXT NOT NULL, path TEXT NOT NULL,
                expires_utc INTEGER NOT NULL, secure INTEGER NOT NULL,
                httponly INTEGER NOT NULL, last_access_utc INTEGER NOT NULL,
                has_expires INTEGER NOT NULL DEFAULT 1,
                persistent INTEGER NOT NULL DEFAULT 1,
                priority INTEGER NOT NULL DEFAULT 1,
                encrypted_value BLOB DEFAULT '');
```

In the following, we show the records in each of these tables.

```
select * from cookies;
13067963800050281|.google.com|PREF|ID=1271b0ef863f405f:TM=1423490200:
LM=1423490200:S=CvT6gX-mx_tpirdd|/|13131035800050281|0|0|
130684849878725 48|1|1|1|

select * from meta;
version|7
last_compatible_version|5
```

Due to the use of these cookies, GSB has been criticized ever since web browsers have started to use them (see [Ken14]). The cookie is allegedly used by the US National Security Agency to identify individual computers [SPG13]. To these criticisms, Google responded that the cookies were not used to track users but only to monitor the performance of the service on the server-side and to catch bugs.[4] However, since they are needed by Google to operate SB, the browsers can not disable it. Nevertheless, Chromium and Firefox have decided to isolate the SB cookie from the others with the purpose of achieving maximum privacy.[5]

6.3.5 Lookup API versus Safe Browsing API

The Lookup API was designed to cater to the needs of some specific applications. It is in fact most suitable for applications that simply want to query the state of URLs and do not mind sending the URLs to Google. The Lookup API comes with the important advantage that it is straightforward and easy to implement for developers. Indeed, clients only need to make an HTTP GET or POST request with the URLs. The Lookup API however has drawbacks in terms of privacy and performance. As URLs are sent in clear to the servers, the privacy of users is at stake as the SB server can build the complete browsing history of a client. Furthermore, each request implies latency due to the network round-trip.

Safe Browsing API is more privacy friendly as API users exchange data with the server using hashed URLs. Another reason being that it requires less frequent access to the server thanks to the local database. Indeed, the local copy makes the Safe Browsing API excel in performance as the browser does not need to query the server every time it needs to check a URL. This reduces the response time for deciding whether a target URL is malicious or not. The major drawback of the Safe Browsing API is its implementation complexity. Safe Browsing API users need to be aware of the internal structure of how the server stores hashed URLs in the blacklists, and implement the canonicalization procedure. Additionally, they have to periodically update their local database. If there are updates, they also need to download the new lists of hashed prefixes. To decide whether a URL is malicious they are also required to download and compare the full hash value of URLs that are hit in the local database.

6.4 Facilitating Safe Browsing Usage

Google also provides two satellite tools to complement GSB. One of these tools is the *Safe Browsing Diagnostic Tool*. It allows users and webmasters to obtain detailed information about the status of a web page. The other tool allows users to report malicious links or false positives. Below, these tools are described in further detail.

[4] https://code.google.com/p/chromium/issues/detail?id=103243
[5] https://bugzilla.mozilla.org/show_bug.cgi?id=368255

6.4.1 Safe Browsing Diagnostic Tool

In order to use the SB service, a potential client has to obtain an API key. While the procedure to obtain a key is simple, it nevertheless requires registering for a GOOGLE account even when the client simply wishes to know the status of a single web page. Hence, to ease access to the service, GOOGLE provides a Safe Browsing Diagnostic tool[6] that returns the status of a web page without requiring clients to obtain an API key.

In addition to reporting a web page as malicious (as in the interstitial warning page), the tool also provides details on why the web page was flagged as malicious and on how attackers could successfully obfuscate or inject malicious payloads in the web page. The diagnostic page provides detailed information about GOOGLE's automatic investigations and findings and hence allows webmasters to identify weak and vulnerable links in the network. The tool can also be accessed via the interstitial warning page by clicking the button with the tag "**Why this page was blocked?**" (see Figure 6.5).

The diagnostic page returns the following information about a queried web page:

- The current listing status of a site and information on how often a site or parts of it were listed as malicious in the past.
- The last time GOOGLE analyzed the page, when it was last malicious, what kind of malware GOOGLE encountered and so forth.
- Did the site facilitate the distribution of malicious software in the past?
- Has the site hosted malicious software in the past? Further information on the victim sites that initiated the distribution of malicious software.

Figure 6.9 presents the result of a query to the Safe Browsing diagnostic server for `ianfette.org`. The result shows that "*some pages on this website install malware on visitors' computers*".

Figure 6.10 presents the result of a query for the application server that hosts `ianfette.org`. Along with `ianfette.org`, three other websites on this network, namely `webonomia.com`, `600hudsonst.com` and `andcoastalhillsdermatology.com` were found to be dangerous over the last ninety days.

We note that while the diagnostic tool is very useful for webmasters, it may however allow attackers or malicious entities to either test the effectiveness and reach of their malware or to surreptitiously learn weak links in a network.

6.4.2 Reporting Tools

GSB also provides tools for a web administrator or an SB client to report incorrect data. A user can report phishing URLs that are not currently on the list using the link: `https://www.google.com/safebrowsing/report_phish/?hl=en`. They may also report for false positives (URLs not actually a phishing link) using the link: `https://www.google.com/safebrowsing/report_error/?hl=en`. In order to report malware

[6] `https://www.google.com/transparencyreport/safebrowsing/diagnostic/?hl=en`

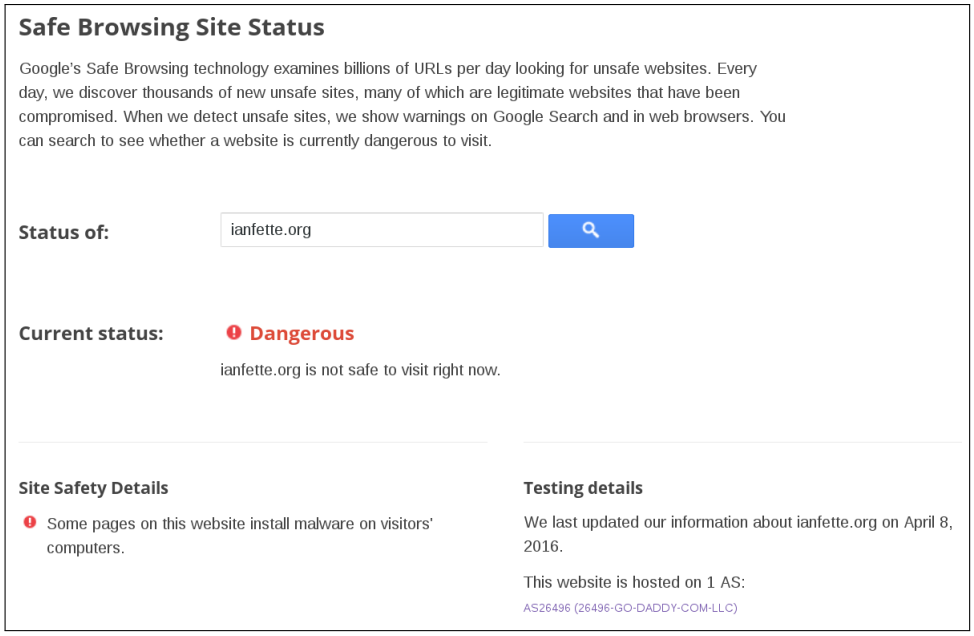

Figure 6.9: Diagnostic result for ianfette.org.

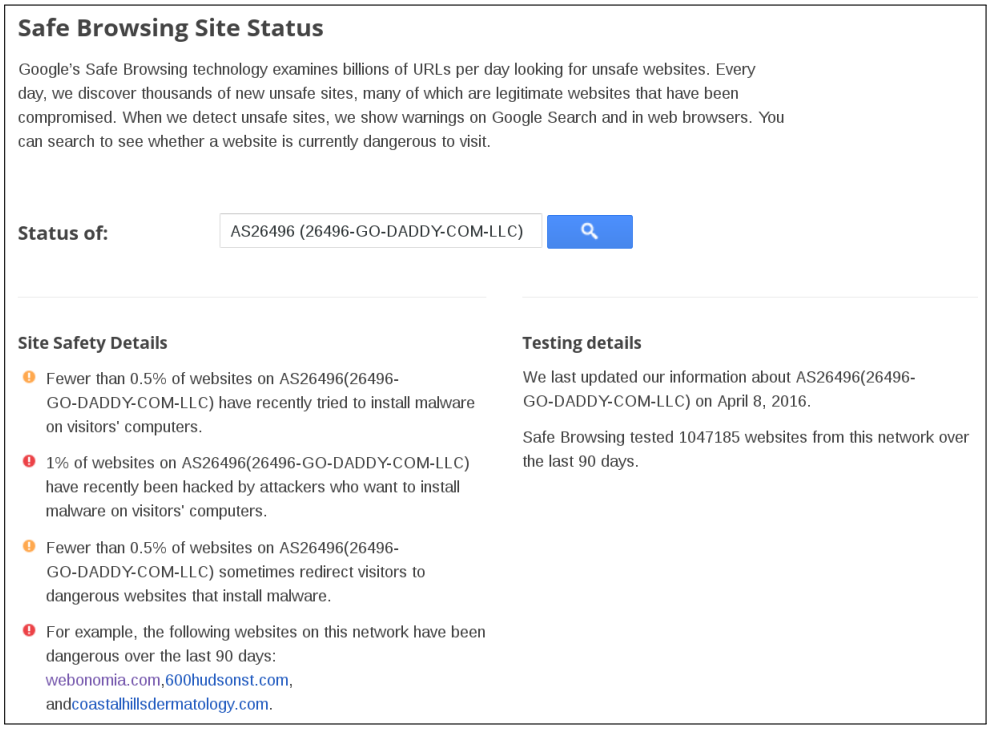

Figure 6.10: Diagnostic result for the application server that hosts ianfette.org.

URLs that are not currently on the malware list, users have to follow https://www.google.com/safebrowsing/report_badware/, while for false positives they must report to http://www.stopbadware.org/home/reviewinfo. Upon receipt of the report, Google analyzes the reported link and takes the needful actions.

6.5 Effectiveness and Usability Studies

Due to the popularity of its SB services, Google Safe Browsing has attracted the attention of the research community to study the usability and effectiveness of the tool in combating online threats. Li *et al.* [LHK+16] study the effectiveness of combinations of browser, search, and direct webmaster notifications at reducing the duration a site remains compromised. The authors capture the life cycle of 760,935 hijacking incidents from July, 2014-June, 2015 and observe that direct communication with webmasters through SB tools increases the likelihood of cleanup by over 50% and reduces infection period by at least 62%.

Akhawe and Felt in [AF13] use Mozilla Firefox and Google Chrome's in-browser telemetry to observe over 25 million warning impressions. The authors observed that, users continued through a tenth of Mozilla Firefox's malware and phishing warnings and through a quarter of Google Chrome's malware and phishing warnings. The work clearly demonstrates that security warnings can be effective in practice and hence security experts and system architects should not ignore the benefits of communicating security information to end users.

Almuhimedi *et al.* [AFRC14] study the factors that may contribute to why people ignore malware warnings. Through several field experiments, authors conclude that users consistently heed warnings about websites that they have not visited before. However, users respond unpredictably to warnings about websites that they have previously visited.

Egelman *et al.* [ECH08] study the effectiveness of an interstitial warning page over passive warnings (such as changing color of the address bar) for phishing threats. The authors conduct experiments to simulate a spear phishing attack that exposes participants to browser warnings. It was observed that when presented with active warnings, 79% of participants heeded them, which was not the case for the passive warnings where only one participant heeded the warnings.

The work by Egelman and Schechter [ES13] studies how the choice of background color in the warning page and the text describing the recommended course of action impact a user's decision to comply with the warning. They observed that both the text and the background color had a significant effect on the amount of time users spend viewing a warning, however they observed no significant differences with regard to their decisions to ultimately obey that warning.

In the following section, we briefly discuss another SB service due to Yandex.

6.6 Yandex Safe Browsing

Yandex Safe Browsing (YSB) comes in the form of an API [Yan16], and also as a security feature in its browser called *Yandex.Browser*. The Yandex Safe Browsing API is compatible with C#, Python and PHP and is a verbatim copy of Google Safe Browsing API with the only difference that in addition to the phishing and the malware lists provided by Google, the YSB API also includes 17 other blacklists. Each of these lists contains

malicious or unsafe links of a certain category.

Table 6.3: Yandex blacklists. Information could not be obtained for cells marked with *.

List name	Description	#Entries
goog-malware-shavar	malware	283,211
goog-mobile-only-malware-shavar	mobile malware	2,107
goog-phish-shavar	phishing	31,593
ydx-adult-shavar	adult website	434
ydx-adult-testing-shavar	test file	535
ydx-imgs-shavar	malicious image	0
ydx-malware-shavar	malware	283,211
ydx-mitb-masks-shavar	man-in-the-browser	87
ydx-mobile-only-malware-shavar	malware	2,107
ydx-phish-shavar	phishing	31,593
ydx-porno-hosts-top-shavar	pornography	99,990
ydx-sms-fraud-shavar	sms fraud	10,609
ydx-test-shavar	test file	0
ydx-yellow-shavar	shocking content	209
ydx-yellow-testing-shavar	test file	370
ydx-badcrxids-digestvar	.crx file ids	*
ydx-badbin-digestvar	malicious binary	*
ydx-mitb-uids	man-in-the-browser android app UID	*
ydx-badcrxids-testing-digestvar	test file	*

Table 6.3 provides the name and description of the blacklists with the number of prefixes present in each. We highlight that the malware lists `goog-malware-shavar` and `ydx-malware-shavar` are identical. The same holds for the malware lists meant for mobile devices: `goog-mobile-only-malware-shavar` and `ydx-mobile-only-malware-shavar`, and for the phishing lists `goog-phish-shavar` and `ydx-phish-shavar` respectively. By comparing the `goog-malware-shavar` lists of Google and Yandex, we observed that there are only 36,547 prefixes in common. Similarly, the lists `googpub-phish-shavar` and `goog-phish-shavar` of Google and Yandex respectively have only 195 prefixes in common. We argue that these anomalies exist because the lists might not be up-to-date on the Yandex server.

6.7 Summary

SB services are valuable tools to fight online phishing and malware threats. Due to the increasing number of such threats, even independent web service providers such as Twitter and Bitly also employ GSB or GSB like services to prevent users from disseminating malicious URLs. Facebook has developed a phishing and malware filter called *Link Shim* [Fac12] that extensively relies on GSB. Figure 6.11 provides a schema that represents the major actors in the SB ecosystem.

All the services described in this chapter (other than GSB and YSB) are privacy unfriendly by design: either the URL or a part of it is sent in clear to the servers. To our knowledge, Google and thereby Yandex provide the only SB services with built-in

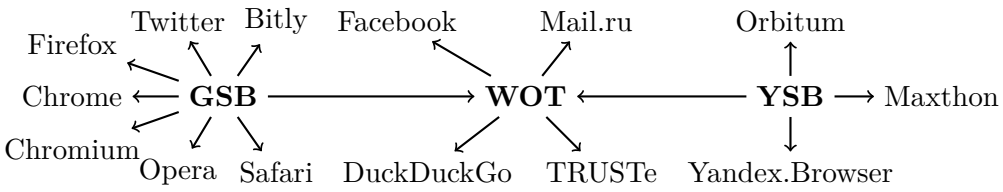

Figure 6.11: The Safe Browsing ecosystem.

privacy features. While Yandex does not provide a Lookup API, Google continues to maintain the same. However, Google recommends the following on the usage of its APIs:

> "*If you are not too concerned about the privacy of the queried URLs, and you can tolerate the latency induced by a network request, consider using the Safe Browsing Lookup API since it's much easier to implement. Otherwise, the Safe Browsing API v3 may be a better choice for you.* "

Chapter 7

Privacy Analysis of Safe Browsing

Contents

7.1 Introduction **125**
7.2 Single Prefix Hit **128**
7.2.1 Analysis 129
7.3 Multiple Prefix Hits **130**
7.3.1 Collisions on Two Prefixes 131
7.3.2 URL Re-identification 132
7.3.3 Statistics on Decompositions 134
7.3.4 A Tracking System based on Safe Browsing 137
7.4 Blacklist Analysis **139**
7.4.1 Inverting Digests 139
7.4.2 Orphan Prefixes 140
7.4.3 Presence of Multiple Prefixes 141
7.5 Mitigations **142**
7.6 Summary **143**

7.1 Introduction

Any SB service can be modeled as a client who wishes to query a database of malicious URLs stored on an SB server. As we have seen in the previous chapter, the Lookup API solves this problem but gives no privacy to users as each query to the server reveals the visited URL. In order to have a privacy friendly SB service, one would ideally wish to execute a private membership protocol between the client and the SB server. From the results of Chapter 4, if we consider the database to be public, such a protocol can be conceived using a variant of PIR. Figure 7.1 presents a schematic representation of an SB service based on a PIR. However, as we have seen in Chapter 4, such a solution is not scalable.

Google's approach to Safe Browsing (GSB v3) can be seen as a solution which lies between the privacy unfriendly solution of Lookup API and the PIR-based privacy-preserving

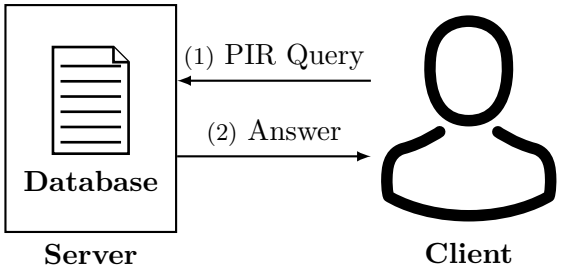

Figure 7.1: Safe Browsing à la PIR.

solution. The underlying objective is to obtain the best of both worlds namely, privacy and scalability. GSB v3 can be simplified as in Figure 7.2, where the simple idea of adding a local database on the client's side reduces the privacy risks as the number of requests made to the server gets drastically reduced. This is because the client now queries the server only when a URL creates a hit in the local copy. However, information relative to a visited URL is still leaked when the client needs to contact the server.

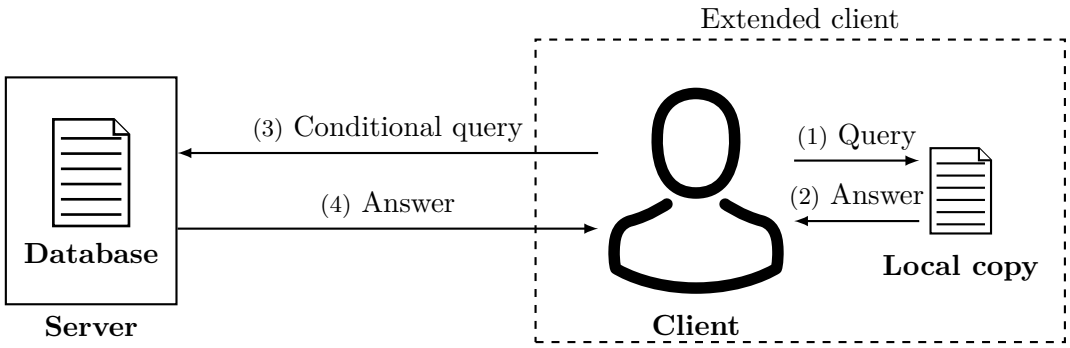

Figure 7.2: A simplified structure of Google Safe Browsing. The client makes a conditional query to the server when the URL creates a hit in the local copy.

In order for this information leakage to be minimal, GSB (and therefore YSB) employs a query anonymization technique that can be seen as a combination of *hashing* and *truncation*. Hashing is used to create pseudonyms for URLs. Generating pseudonyms (digests) for URLs however does not suffice to anonymize the data, and hence pseudonyms are truncated to create collisions. Truncation ensures that several URLs share the same reduced pseudonym (prefix) and hence creates anonymity on the request.

The above anonymity argument is valid as long as the requests sent to the server are random and therefore lack any correlation. Now, let us consider a situation where requests can in fact be correlated. We note that correlation between requests can be either on the part of the client or the server. A client unintentionally injects correlation between requests when it sends multiple prefixes for a given URL. This happens when several decompositions of a given URL create hits in the local database. On the other hand, a server may deliberately create correlation when it aggregates requests received over time, *i.e.*, when it creates temporal correlation.

Motivation. A malicious server may now wish to de-anonymize the requests and learn about the visited URLs. Clearly, de-anonymizing requests in a Lookup API is trivial and requires no effort. This is also the case for most SB tools such as WOT. A Lookup-like API allows the SB provider to reconstruct completely or partly the browsing history of a client from the data sent to the servers. However, in case of GSB v3, a full reconstruction of the browsing history may not seem to be feasible. In face with this, SB providers may instead wish to learn if a client has visited some selected web pages. By associating traits to pages, the ultimate goal of the SB provider could be to detect users' behavior such as political opinions, sexual orientation or allegiance to terrorist groups. As SB services are included in almost all popular web browsers, a malicious SB service may also prove to be a handy tweak for Google and Yandex to turn all SB-enabled browsers into a tracking tool of their own. The threat can have a wide reach because of the fact that GSB is open-source. Being open-source allows any third party to clone it (as currently done by Yandex) and offer a similar service. If the third party is malicious, then it can abusively use GSB for tracking.

Google Chrome Privacy Notice [Goo14b] includes a section on GSB. It states that:

> "*Google cannot determine the real URL from this information* (read prefix)".

Google reiterates this statement in a document concerning the GSB usage in Mozilla Firefox [Goo12]. These guarantees have evidently allowed GSB to be massively used by end users and even by other web service providers (such as Facebook and Twitter). Apart from these statements, there is no other privacy analysis of GSB. The goal of this chapter is to provide an independent privacy analysis of GSB and its sibling YSB.

Technical roadmap and contributions. De-anonymization attacks often rely on the amount of information that an adversary can gather. Hence, any privacy analysis should take into account the quantity of available information. In case of SB, information can be quantified by the number of prefixes sent to the server for a given URL. In situations where a single prefix for a URL is sent to the GSB server, we employ a simple *balls-into-bins* argument to estimate the anonymity set size for the prefix, *i.e.*, the number of URLs that generate the same prefix (Section 7.2).

However, in cases when the client sends multiple prefixes for a given URL, the analysis becomes more involved. In theory, distinct URLs can also share common multiple prefixes in their decompositions and hence may provide anonymity. We characterize the different cases for such multiple collisions and collect their statistics on URLs using the *Common Crawl* dataset [Com15]. Our experimental analysis estimates the rate of such collisions and shows that hashing and truncation fail to prevent re-identification of small-sized domains or certain URLs of larger domains. We further materialize this in the form of an algorithm that SB providers could potentially employ for tracking (Section 7.3).

The analysis of the databases of malicious prefixes provided by Google and Yandex reveals several URLs which have multiple prefixes included in the databases. These provide concrete examples of URLs and domains that can be easily tracked. By crawling their databases, we further detect a number of "suspicious" prefixes that we call *orphans*.

Orphans trigger communication with the servers, but no full digest corresponds to them (Section 7.4).

Motivated by our findings, we discuss and evaluate a privacy enhancing technique to reduce information leakage while maintaining the current architecture. Our proposal is efficient and can be directly integrated into SB clients (Section 7.5).

In the following section we present an analysis of the case when a single prefix for a URL creates a hit in the local database. Studying this case allows us to know if a user's browsing history can be fully/partly constructed.

7.2 Single Prefix Hit

A simple way to understand GSB and YSB consists in considering them as a probabilistic test run by the client to filter malicious URLs. Whenever, the test executed by the browser is positive, Google or Yandex servers are contacted to remove any ambiguity. While a negative test leaks no information about the URL to Google and Yandex, a positive test sends prefix(es) of certain decompositions of the target URL. Table 7.1 summarizes all the cases with respect to two factors: existence in the prefix lists and if a connection with the distant server is required. The exact number of prefixes sent depends on the number of hits in the local database.

Table 7.1: Events leaking information in GSB and YSB.

Event	Local hit	Connection
True positive	Yes	Yes
True negative	No	No
False positive	Yes	Yes
False negative	No	No

Let us consider the following URL: `https://conf.org/2016/cfp.html`. Its decompositions are shown in Table 7.2. Let us further assume that the value `0x24e04dde` is present in one of the blacklists.

Table 7.2: Decompositions of `https://conf.org/2016/cfp.html`.

URL	32-bit prefix
`conf.org/2016/cfp.html`	`0x24e04dde`
`conf.org/2016/`	`0xf9aef594`
`conf.org/`	`0xed37d926`

Now, let us consider a situation where a client visits a web page that creates a hit on the prefix `0x24e04dde`. The client then sends the prefix to the server. The server's goal is to re-identify the visited URL from the received prefix. In order to determine if the prefix is enough for Google or Yandex to re-identify the corresponding URL, we need to measure the related uncertainty in re-identification. Hence, the privacy metric that we consider measures the number of URLs that share a given prefix. This metric is

the *anonymity set size* of Chapter 3 §3.5. The higher is the value of the metric, the more difficult is the re-identification and hence better is the privacy achieved. The metric yields a simple yet reliable method to analyze and quantify the information leakage through prefixes.

7.2.1 Analysis

One may argue that there are infinite number of pre-images for a 32-bit prefix, hence the privacy metric that estimates the uncertainty in re-identification should be infinitely large. However, the crucial point here is that the total number of URLs on the web is finite and hence the privacy metric can at most be finitely small.

The privacy metric can be easily estimated using the probabilistic model of *balls-into-bins*. In the SB context, prefixes represent the bins and URLs can be considered as balls. We are interested in the maximum, minimum and the average value that the privacy metric can take. These values respectively measure the worst-case, best-case and average-case uncertainty for re-identification. Let us suppose m to be the number of balls and n to be the number of bins. Then according to the results from Chapter 3 §3.2.5, the average number of balls in any bin is $\frac{m}{n}$. We further note that, according to the result of Ercal-Ozkaya [EO08] (*Cf.* Theorem 3.2.2 from Chapter 3 §3.2.5), for a constant $c > 1$, and $m \geq cn \log n$, the minimum number of balls in any bin is $\Theta\left(\frac{m}{n}\right)$. As a result, the minimum value of the metric should not deviate too much from the average value and hence to avoid redundancy, we do not consider it in our analysis. Finally, in order to compute the maximum, we use Theorem 3.2.1 from Chapter 3 §3.2.5.

In 2008, Google [Goo08] claimed to know 1 trillion unique URLs. Since then, Google has reported 30 trillion URLs in 2012 and 60 trillion in 2013. These data are summarized in the first two rows of Table 7.3. The table also presents the number of domain names recorded by verisign [Ver15]. Using the results on balls-into-bins, and the provided Internet data, we compute the maximum and the average value of the metric for unique URLs and domain names. Results are provided for different prefix sizes in Table 7.3. When GSB was started in 2008, at most 443 URLs matched a given 32-bit prefix. It has increased over the years to reach 14,757 in 2013. Even in the average case, it is hard for Google and Yandex to re-identify a URL from a single 32-bit prefix. The case of domain names is slightly different because the space of domain names is much smaller and its dynamic is far slower than the one of URLs. In the worst case, two domain names will collide to the same prefix. Domain names can hence be re-identified with high certainty. However, the server does not know if the received prefix corresponds to a domain name or to a URL, hence the ambiguity can not be resolved.

To conclude this analysis, a single prefix per URL does not allow the SB server to reconstruct the browsing history of the client. So far, the solution seems to be privacy-preserving as long as the client only reveals a single prefix.

Table 7.3: Max. and avg. values for URLs and domains with prefix size ℓ. 0^* represents a value close to 0.

	URLs (10^{12})			**Domains** (10^6)		
Year	2008	2012	2013	2008	2012	2013
Number	1	30	60	177	252	271
ℓ (bits)	max, avg			max, avg		
16	$2^{28}, 2^{23}$	$2^{28}, 2^{28}$	$2^{29}, 2^{29}$	3101, 2700	4196, 3845	4498, 4135
32	443, 232	7541, 6984	14757, 13969	2, 0.04	3, 0.05	3, 0.06
64	2, 0^*	2, 0^*	2, 0^*	1, 0^*	1, 0^*	1, 0^*
96	1, 0^*	1, 0^*	1, 0^*	1, 0^*	1, 0^*	1, 0^*

7.3 Multiple Prefix Hits

In this section, we analyze the case when SB servers include multiple prefixes for a domain in the local database. We later see in Section 7.4 that it is also possible that the SB servers may flag several decompositions of a URL as malicious and hence include the respective prefixes in the local database. Let us first understand why the local database may contain several prefixes for a domain. In fact, it becomes necessary when a domain has a subset of sub-domains and URL paths which host several malicious URLs. Then, the sub-domains and the paths can be blacklisted instead of including each malicious URL in the database. This approach saves memory footprint on the client's side. We note that one could possibly include only the prefix of the domain to blacklist all its malicious sub-domains and paths. However, this approach also blacklists all non-malicious URLs on the domain. Whence, multiple prefixes are indispensable to prevent certain URLs from being flagged as malicious. Figure 7.3 presents a simple example for a domain `b.c`, where the colored nodes (for `d.b.c`, `a.b.c/1` and `a.b.c/2`) represent the malicious resources. A naive way to flag the colored nodes as malicious would be to include the corresponding prefixes in the local database. However, as `a.b.c` has only two web pages and both being malicious, therefore, `a.b.c` can itself be flagged as malicious instead of flagging the two web pages that it hosts. Also, `d.b.c` must be flagged as malicious. Hence, in total only two prefixes are required to be added to the blacklists instead of three. We note that if the entire domain, `b.c` is blacklisted instead, then it will also blacklist the benign sub-domain `e.b.c`.

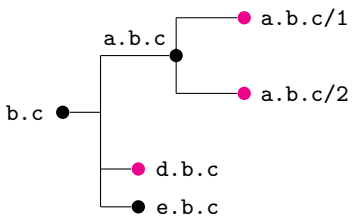

Figure 7.3: A sample domain hierarchy for `b.c`. Colored nodes represent malicious resources. A real world example can be `google.com` for `b.c`; `mail.google.com` for `a.b.c`, `analytics.google.com` for `d.b.c`, and `maps.google.com` for `e.b.c`.

We note that it is also possible that multiple decompositions for a URL create hits in the local database. This could be because of accidental collisions on 32-bit prefixes. Since multiple prefix hits require the client to send all the prefixes to the server, this may potentially leak user's privacy. The essential reason being that multiple prefixes force the client to send more information to the server than in the case of a single prefix hit. Clearly, the amount of information on the URL obtained is proportional to the actual number of prefixes received.

In fact, users' privacy is still at risk even when a URL does not create multiple hits in the local database. Indeed, the server can still re-identity the URL by aggregating requests sent by the client and exploiting the temporal correlation between the queries. YANDEX and GOOGLE can identify the requests of a given user thanks to the SB cookie. Continuing with our previous example URL, a user visiting: `conf.org/2016/cfp.html` (with prefix `0x24e04dde`) is very likely to visit the submission website: `conf.org/2016/submission/`(with prefix `0x51644dab`). Instead of looking at a single query, the SB server now needs to correlate two queries. A user making two queries for the prefixes `0x24e04dde` and `0x51644dab` in a short period of time is planning to submit a paper.

In the following, we present an analysis of multiple prefix hits for a URL. For the sake of clarity and comprehensibility of the privacy analysis, we henceforth consider the simplified case of 2 prefixes. The analysis for the general case follows in a straightforward manner.

7.3.1 Collisions on Two Prefixes

As in the single prefix case, more than two distinct URLs may yield the same two prefixes. The larger is the number of such URLs, the more difficult is the re-identification. These URLs exist due to three possible types of collisions on 32-bit prefixes. In order to illustrate the different possible collisions, we present a set of examples in Table 7.4. We assume that the client visits the target URL `a.b.c` and hence the server receives the corresponding two prefixes, denoted by A and B. The server using these prefixes must determine the exact URL visited by the client. The next 3 URLs exemplify the different collisions. In the first type (Type I), several distinct yet "related" URLs share common decompositions and these decompositions yield the shared prefixes. We note that 2 distinct URLs are "related" if they have common sub-domains. The second type of collisions (Type II) is due to distinct yet "related" URLs that share one decomposition and hence one common prefix, while the other common prefix is due to the collision on truncated digests. Finally, the last type of collisions (Type III) appears when completely "unrelated" URLs generate the same prefixes. The latter may occur again due to collisions on the truncated digests. In the following, by a Type I URL, we mean a URL that generates a Type I collision with a given URL. We similarly define Type II and Type III URLs for a given URL.

Clearly, $\mathbb{P}[\text{Type I}] > \mathbb{P}[\text{Type II}] > \mathbb{P}[\text{Type III}]$, where $\mathbb{P}[X]$ denotes the probability of an event X. Under the uniformity assumption of hash functions, a Type III collision is highly unlikely, with a probability of $\frac{1}{2^{64}}$. We note that for Type I and Type II collisions to occur, the URLs must share at least one common decomposition. The probability of

Table 7.4: An example with different possible collisions. The decomposition algorithm is the one described in Chapter 6. URLs `g.a.b.c` and `a.b.c` are "related" since they share two common decompositions, namely `a.b.c/` and `b.c./`.

		URL	Decomposition	Prefixes
	Target URL	`a.b.c`	`a.b.c/`	A
			`b.c/`	B
Coll. Type	Type I	`g.a.b.c`	`g.a.b.c/`	C
			`a.b.c/`	A
			`b.c/`	B
	Type II	`g.b.c`	`g.b.c/`	A
			`b.c/`	B
	Type III	`d.e.f`	`d.e.f/`	A
			`e.f/`	B

these collisions hence depends on the number of decompositions of URLs hosted on the domain. In general, the smaller is the number of decompositions per URL, the lower is the probability that Type I and Type II URLs exist. Moreover, a Type II URL exists only if the number of decompositions on a domain is larger than 2^{32}. We later show that no Type II URL exists by empirically estimating the distribution of decompositions over domains. As a result, the ambiguity in the re-identification can only arise due to Type I collisions. In the following, we discuss the problem of URL re-identification with a focus on URLs that admit Type I collisions.

7.3.2 URL Re-identification

We note that a target URL with few decompositions has a very low probability to yield Type I collisions, and hence it can be easily re-identified. In case of URLs with a large number of decompositions, the server would require more than 2 prefixes per URL to remove the ambiguity. Nevertheless, the SB provider can still determine the common sub-domain visited by the client using only 2 prefixes. This information may often suffice to identify suspicious behavior when the domain in question pertains to specific traits such as pedophilia or terrorism. It is pertinent to highlight that the SB service provided by WOT collects the domains visited by its clients [WOT15a]. Hence, in the scenario where GSB and YSB servers receive multiple prefixes for a URL, the privacy achieved is the same as that ensured by services such as WOT.

Now, let us further analyze the problem of re-identifying URLs for which Type I collisions occur. To this end, we consider an illustrative example of a domain `b.c` that hosts a URL `a.b.c/1` and its decompositions (see Table 7.5). We assume that these are the only URLs on the domain. The URL generates four decompositions. Two of these decompositions include the domain name `'a'` as a sub-domain while the remaining two do not. These decompositions yield prefixes denoted by A, B, C and D respectively. We note that the most general canonicalized URL as defined in Chapter 6 is `http://a.b.c/1/2.ext?param=1`. Hence, the considered URL is only a slightly simplified form of the most general URL, where the query part of the URL has been removed. Therefore, it largely

represents all the canonicalized URLs.

Table 7.5: A sample URL on `b.c` with its 4 decompositions.

URL	Decompositions	Prefix
`a.b.c/1`	`a.b.c/1`	A
	`a.b.c/`	B
	`b.c/1`	C
	`b.c/`	D

We analyze the following three cases depending on the prefixes sent by the client to the SB server:

- **Case 1.** (A, B) generates hits: If the server receives these prefixes, it can be sure that the client has visited the URL that corresponds to the first prefix A, *i.e.*, `a.b.c/1`. This is because `a.b.c/1` is the only URL that generates two decompositions yielding prefixes A and B. For instance, the decompositions of the URL `a.b.c/` can only generate prefixes B, C or D but not A.

  The probability that re-identification fails in this case is $\mathbb{P}[\text{Type III}] = \frac{1}{2^{64}}$. This holds because we assume that the domain `b.c` hosts only 4 URLs, hence the probability that the re-identification fails is the same as the probability of finding a Type III URL for prefixes (A, B). Our assumption ensures that no Type II URLs exist.

- **Case 2.** (C, D) generates hits: In this case, the possible URLs that the client could have visited are: `a.b.c/1`, `a.b.c/` or `b.c/1`. This is because these are the only URLs which upon decomposition yield both C and D. These URLs correspond to prefixes A, B and C respectively. Hence, in order to remove the ambiguity and re-identify the exact URL visited by the client, the SB provider would include additional prefixes in the local database. If it includes the prefix A, in addition to C and D, then it can learn whether the client visited the URL `a.b.c/1` or `b.c/1`. More precisely, if the client visits `a.b.c/1` then prefixes A, C and D will be sent to the server, while if the client visits `b.c/1`, then only C and D will be sent. Similarly, in order to distinguish whether the client visits `a.b.c/` or `b.c/`, the SB provider would additionally include the prefix B.

- **Case 3.** One of $\{A, B\} \times \{C, D\}$ generates hits: If the prefix A creates a hit, then the visited URL is `a.b.c/1`. This is because `a.b.c/1` is the only URL that upon decomposition can yield the prefix A. If the prefix B creates a hit, then the client has either visited `a.b.c/1` or `a.b.c/`. This is again because these are the only two URLs capable of generating the prefix B. As in Case 2, in order to distinguish between these two URLs, the SB provider is required to include an additional prefix A in the prefix database.

As a general rule, all the decompositions that appear before the first prefix are possible candidates for re-identification. Consequently, lower-level domain names and URL paths can be re-identified with a higher certainty than the ones at a higher level. To this end,

we consider the case of *leaf* URLs on a domain. We call a URL on a given domain a *leaf*, if it does not belong to the set of decompositions of any other URL hosted on the domain. A leaf URL can also be identified as a leaf node in the domain hierarchy (see Figure 7.4). Type I collisions for these URLs can be easily eliminated during re-identification with the help of only two prefixes. The first prefix corresponds to that of the URL itself, while the other one may arbitrarily correspond to any of its decompositions. In the example of Table 7.5, the URL `a.b.c/1` is a leaf URL on the domain `b.c`, and hence it can be re-identified using prefixes (A, B).

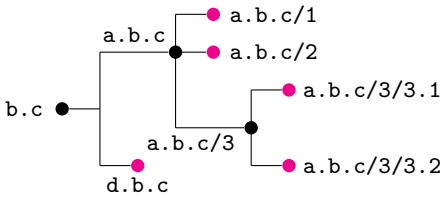

Figure 7.4: A sample domain hierarchy for `b.c`. Colored nodes are leaf URLs: `a.b.c/1`, `a.b.c/2`, `a.b.c/3/3.1`, `a.b.c/3/3.2` and `d.b.c`. A real world example can be `google.com` for `b.c`; `mail.google.com` for `a.b.c` and `analytics.google.com` for `d.b.c`.

Clearly, only non-leaf URLs contribute to Type I collisions. Hence, in order to re-identify non-leaf nodes, one must include more than 2 prefixes per node. Our observations further raise the question of the distribution of decompositions over domains that we explore in the following section.

7.3.3 Statistics on Decompositions

In the following, we support our analysis of the previous section using extensive experiments to estimate the distribution of URLs, decompositions and collisions over domains.

Our experiments have been performed on the web corpus provided by the *Common Crawl* project [Com15]. Common Crawl is an open repository of web crawl data collected over the last 7 years (the project was started in 2009). It contains raw web page data, metadata and text extractions. For our experiments, we have used the corpus of April 2015. This crawl archive is over 168 TB in size and holds more than 2.11 billion web pages. We note that the data included in Common Crawl is not exact. For instance, the maximum number of URLs hosted on a domain is of the order of 10^5. However, there are several domains such as `wikipedia.org` which host over a billion URLs. This peak is due to the fact that crawlers do not systematically collect more pages per site than this bound due to limitations imposed by the server. One may also find small-sized domains for which the crawl archive does not include all web page data, in particular all the URLs on the domain. Despite this bias, the corpus allows us to obtain a global estimate of the size of the web and determine the distribution of URLs and decompositions.

It is worth noticing that popular domains often host more URLs than the non-popular ones. This generally implies that the number of unique URL decompositions and eventual collisions on popular domains are larger than those on random/non-popular ones. We hence consider two datasets in our experiments. Our first dataset contains web pages on

the 1 million most popular domains of Alexa [Ale15]. We also collected 1 million random domains from Common crawl and then recovered web pages hosted on these domains. This forms our second dataset. We note that the random dataset may contain Alexa domains. The number of URLs and the total number of decompositions provided by these datasets is given in Table 7.6. Our dataset on popular domains contains around 1.2 billion URLs, while the one on random domains includes over 427 million URLs. URLs in the Alexa dataset yield around 1.4 billion unique decompositions in total, while the random dataset generates around 1 billion decompositions.

Table 7.6: Our datasets.

Dataset	#Domains	#URLs	#Decompositions
Alexa	1,000,000	1,164,781,417	1,398,540,752
Random	1,000,000	427,675,207	1,020,641,929

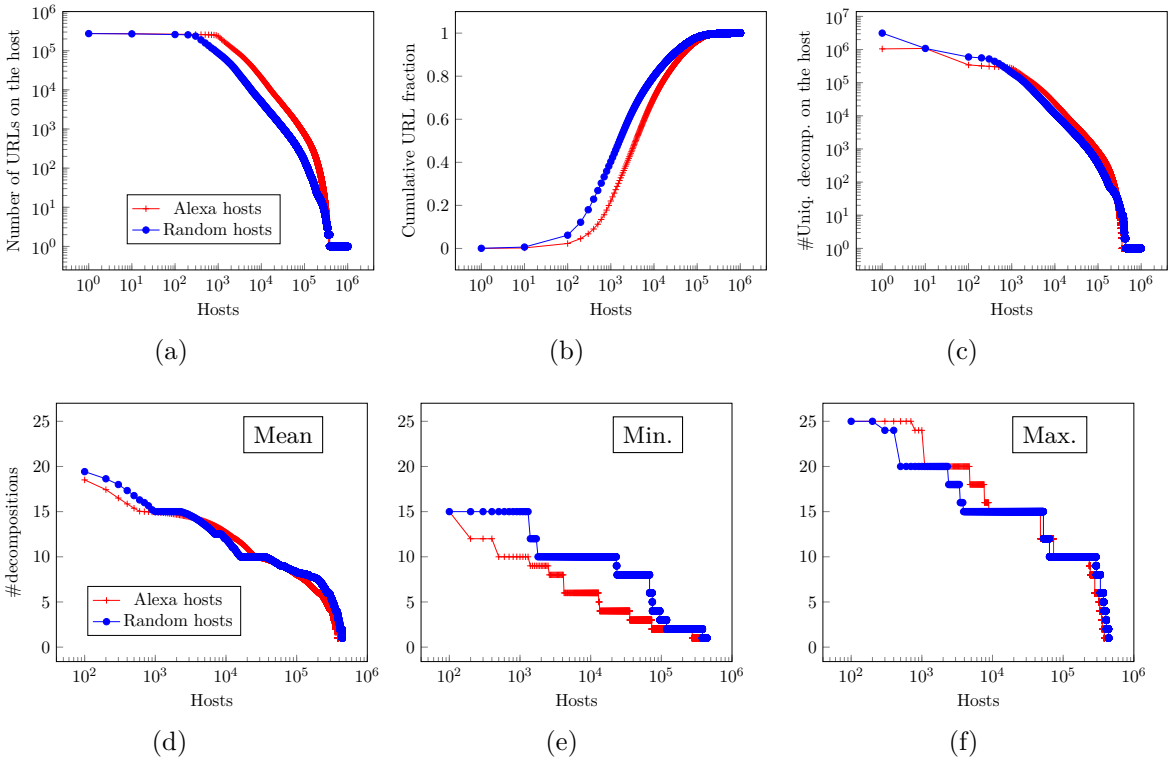

Figure 7.5: Distribution of URLs and decompositions on hosts from the two datasets. Figure (a) presents the distribution of URLs over hosts, while (b) presents its cumulative distribution. (c) shows the distribution of decompositions over hosts. (d), (e) and (f) respectively present the mean, minimum and maximum number of URL decompositions on the hosts.

Figure 7.5a presents the number of URLs hosted on the domains belonging to these two datasets. Clearly, the Alexa domains host larger number of URLs than the random domains. The most number of URLs hosted by a domain from either of the datasets is around 2.7×10^5. We also note that around 61% of the domains in the random dataset are single page domains. Figure 7.5b presents the cumulative fraction of URLs for the two datasets. Our results show that for the Alexa dataset, only 19,000 domains cover 80% of

all the URLs while, for the random dataset, only 10,000 domains span the same percentage of URLs. These results give strong empirical evidence (due to a large and random dataset) to previous results by Huberman and Adamic [HA99] that demonstrate that the number of web pages per site is distributed according to a power law. This implies that on a log-log scale the number of pages per site should fall on a straight line. For the random dataset, we fit a power-law distribution of the form:

$$p(x) = \frac{\alpha - 1}{x_{\min}} \left(\frac{x}{x_{\min}} \right)^{-\alpha},$$

where, $x_{\min} = 1$, which denotes the fact that the minium number of URL on a domain is 1; and the parameter α is estimated as $\hat{\alpha}$:

$$\hat{\alpha} = 1 + n \left(\sum_{i=1}^{n} \ln \frac{x_i}{x_{\min}} \right)^{-1} \approx 1.312,$$

where $\{x_i\}$ are the n data points. In our dataset $n = 10^6$. The standard error of the estimate is given by: $\sigma = \frac{\hat{\alpha}-1}{\sqrt{n}} = 0.0004$.

Figure 7.5c presents the number of unique decompositions per host for the two datasets. These numbers are very close to that of the number of URLs and hence verify a similar power law distribution. The domains that cover the majority of the decompositions contain URLs that are difficult to re-identify (due to Type I collisions). While, the domains representing the tail of the distribution provide URLs which can be re-identified with high certainty using only a few prefixes.

In Figure 7.5d,7.5e,7.5f, we respectively present the mean, minimum and maximum number of decompositions of URLs per domain in the two datasets. We note that 51% of the random domains present a maximum of 10 decompositions for a URL, while the same is true for 41% of the Alexa domains. The minimum value on the other hand is higher for random domains: 71% of the Alexa domains have a minimum of 2 per URL, while the fraction is up to 86% in case of random domains. The average number of decompositions for over 46% of the hosts from the two datasets lies in the interval $[1, 5]$. Hence, a URL on these hosts can generate on an average a maximum of $\binom{5}{2} = 10$ Type I collisions on two prefixes. As a result, URLs on these hosts can be re-identified using only a few prefixes.

We now consider Type II collisions. A Type II collision occurs on URLs that share common decompositions. Thus, in order for a Type II collision to occur, the number of decompositions per domain must be at least 2^{32}. However, the maximum number of decompositions per domain from either of the datasets is of the order of 10^7 (see Figure 7.5c), which is smaller that 2^{32}. This implies that Type II collisions do not occur for any of the hosts in our datasets.

As for Type I collisions, we observed that the number of collisions found was proportional to the number of unique decompositions on the host. On the one hand, we found several domains from both the datasets for which the number of such collisions was as high as 1 million, while on the other hand, we found over a hundred thousand domains for which the number of such collisions was less than 20. Hence, many of the non-leaf

URLs on these hosts can be re-identified by inserting as less as 3 prefixes per URL. Interestingly, we observed that 56% of the domains in the random dataset do not have Type I collisions, while the same is true for around 60% of the domains from the Alexa dataset. Consequently, URLs on these domains can be easily re-identified using only 2 prefixes.

7.3.4 A Tracking System based on Safe Browsing

Our analysis shows that it is possible to re-identify certain URLs whenever multiple prefixes corresponding to them are sent to the servers. Relying on this fact, Google and Yandex could potentially build a tracking system based on GSB and YSB respectively. The robustness of the system would depend on the maximum number of prefixes per URL that they choose to include in the client's database. In the following, we denote this parameter by δ. Clearly, the larger is the δ, the more robust is the tracking tool. We note that its value is however chosen according to the memory constraints on the client's side.

Algorithm 2: Prefixes to track a URL

```
Data: link: a URL to be tracked and a bound δ: max. #prefixes to be included.
Result: A list track-prefixes of prefixes to be included.
decomps, track-prefixes, type1-coll ← [ ]
dom ← get_domain(link)
urls ← get_urls(dom)
for url ∈ urls do
    decomps ← decomps ∪ get_decomps(url)
if |decomps| ≤ 2 then
    for decomp ∈ decomps do
        track-prefixes ← track-prefixes ∪ 32-prefix(SHA-256(decomp))
else
    type1-coll ← get_type1_coll(link)
    common-prefixes ← 32-prefix(SHA-256(dom)) ∪ 32-prefix(SHA-256(link))
    if link is a leaf or |type1-coll| == 0 then
        track-prefixes ← common-prefixes
    else
        if |type1-coll| ≤ δ then
            track-prefixes ← common-prefixes
            for type1-link ∈ type1-coll do
                track-prefixes ← 32-prefix(SHA-256(type1-link)) ∪ track-prefixes
        else
            track-prefixes ← common-prefixes
            /* Only SLD can be tracked.                                  */
```

The tracking system would essentially work as follows. First, Google and Yandex choose a $\delta \geq 2$, and build a shadow database of prefixes corresponding to at most δ decompositions of the targeted URLs. Second, they push those prefixes in the client's database. Google and Yandex can identify individuals (using the SB cookie) each time their servers receive a query with at least two prefixes present in the shadow database.

The crucial point to address is how they may choose the prefixes for a given target

URL. In Algorithm 2, we present a simple procedure to obtain these prefixes for a target URL given a bound δ. The algorithm first identifies the domain that hosts the URL, which in most cases will be a Second-Level Domain (SLD) (see Line 2). Using their indexing capabilities, the SB providers then recover all URLs hosted on the domain and then obtain the set of unique decompositions of the URLs (Line 3-5). Now, if the number of such decompositions is less than 3, then the prefixes to be included for the target URL are those corresponding to these decompositions (Line 6-8). Otherwise, the algorithm determines the number of Type I collisions on the URL (Line 9-20). If no Type I collision exists or if the URL represents a leaf, then only two prefixes suffice to re-identify the URL. One of these prefixes is chosen to be the prefix of the URL itself and the other one can be that of any arbitrary decomposition. We however choose to include the prefix of the domain itself (Line 11-13). If the number of Type I collisions is non-zero but less than or equal to δ, then the prefixes of these Type I URLs are also included (Line 15-18). Finally, if the number of Type I collisions is greater than δ, then the URL cannot be precisely tracked. Nevertheless, including the prefix of the URL and that of its domain allows to re-identify the SLD with precision (Line 19-20). We note that if the prefixes are inserted according to this algorithm, the probability that the re-identification fails is only $\left(\frac{1}{2^{32}}\right)^{\delta}$.

The cost of re-identification on the server side is negligible. This is because the shadow database can be used by the server for later lookups.

To illustrate the algorithm, let us consider the following CFP URL for a conference named PETS: petsymposium.org/2016/cfp.php. The domain hierarchy of petsymposium.org is shown in Figure 7.6.

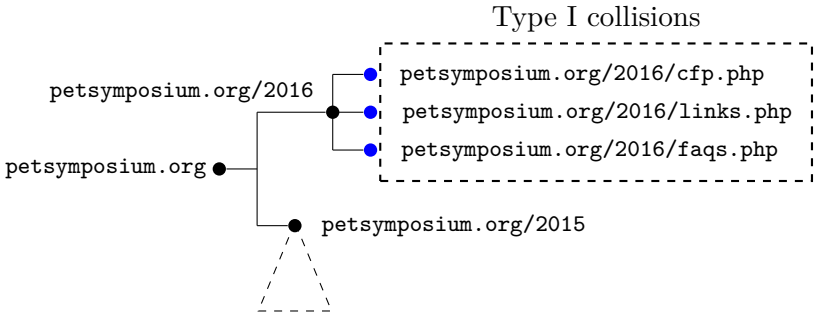

Figure 7.6: The domain hierarchy of petsymposium.org. Leaf URLs are in blue. The dashed triangle represents the sub-hierarchy for petsymposium.org/2015. The URLs in the dashed rectangle create Type I collisions with petsymposium.org/2016.

We first assume that the SB providers wish to identify participants interested in submitting a paper. Hence, the target URL is the CFP URL. The URL has 3 decompositions as given in Table 7.7.

Table 7.7: Decompositions of PETS CFP URL.

URL	32-bit prefix
petsymposium.org/2016/cfp.php	0xe70ee6d1
petsymposium.org/2016/	0x1d13ba6a
petsympsium.org/	0x33a02ef5

Since the target URL is a leaf, prefixes for the first and last decompositions would suffice to track a client visiting the target URL. Now, let us consider the case when the SB provider wishes to track a client's visit on petsymposium.org/2016 web page. The target URL yields Type I collisions with: petsymposium.org/2016/links.php and petsymposium.org/2016/faqs.php. Thus, the SB provider would include the prefixes corresponding to these 2 URLs, that of petsymposium.org/2016 and of the domain petsymposium.org/. In total, only 4 prefixes suffice in this case. Now, whenever the SB server receives the last two prefixes, then it learns that the client has visited the target URL. Additionally, this allows the server to track the other 2 URLs that create Type I collisions.

7.4 Blacklist Analysis

In this section, we present an analysis of the blacklists provided by Google and Yandex. Our analysis has the objective to identify URLs that create multiple prefix hits. These URLs hence provide concrete examples of URLs/domains that can be tracked by the SB providers.

7.4.1 Inverting Digests

As a first step in our analysis, we recover the prefix lists of Google and Yandex. We then use these lists to query their servers using their respective APIs. This allows us to obtain the lists of full digests. Our second step is an attempt to identify the URLs which correspond to these prefixes. To this end, we harvested phishing and malware URLs, domains, and IP addresses from several sources and tested for their belonging to the blacklists of prefixes. The list of all our sources can be found in [Zel15]. We also harvested 1,240,300 malware URLs, 151,331 phishing URLs and 2,488,828 URLs of other categories from BigBlackList [Big15]. Lastly, we obtained 106,923,807 SLDs from the DNS Census 2013 project [Dns13]. The project provides a public dataset of registered domains and DNS records gathered in the years 2012-2013. We included the last dataset to determine the percentage of prefixes in the local database that correspond to SLDs. A description of all the datasets employed in our analysis is given in Table 7.8.

Table 7.8: Dataset used for inverting 32-bit prefixes.

Dataset	Description	#entries
Malware list	malware	1,240,300
Phishing list	phishing	151,331
BigBlackList	malw., phish., porno, others	2,488,828
DNS Census-13	second-level domains	106,923,807

The results of our experiments are shown in Table 7.9. We observe that reconstruction for Google prefixes using Malware, Phishing and BigBlackList datasets is inconclusive: 5.9% for malwares and 0.1% for phishing websites. For Yandex, the situation is better but the majority of the database still remains unknown. However, the DNS Census-13

dataset produces a much larger reconstruction for all the lists except that of the phishing database. The rate is as high as 55% for YANDEX files. We highlight that phishing domains are short-lived and since the DNS Census-13 dataset dates back to 2013, the result of the reconstruction for phishing lists is very limited, only 2.5% for GOOGLE and 5.6% for YANDEX. Nevertheless, these results demonstrate that 20% of the GOOGLE malware list represents SLDs, while 31% of the prefixes in the YANDEX malware lists correspond to SLDs. Relying on our analysis of the previous sections, we may conclude that these prefixes can be re-identified with very high certainty.

It is pertinent to compare the result of our reconstruction with a similar attempt with another list in the past. German censorship federal agency called BPjM maintains a secret list of about 3,000 URLs believed to be unsuitable for women and children. The list is anonymized and distributed in the form of MD5 or SHA-1 hashes as the "BPJM-Modul" [BPj14]. Though similar to the lists handled by GOOGLE and YANDEX, hackers have been able to retrieve 99% of the cleartext entries. We have applied the same approach, yet the reconstruction rate obtained has not been equally high. This proves that in order to reconstruct the database in clear, one would need high crawling capabilities and hence it is not achievable for general users. Furthermore, unlike the BPjM list, the blacklists provided by GSB and YSB are extremely dynamic. This requires a regular crawling of the web, which renders the reconstruction even more difficult.

Table 7.9: Matches found with our datasets.

		#matches (%match)			
	list name	**Malware list**	**Phishing list**	**BigBlackList**	**DNS Census-13**
GOOGLE	goog-malware-shavar	18785 (5.9)	351 (0.1)	6208 (1.9)	63271 (20)
	googpub-phish-shavar	632 (0.2)	11155 (3.5)	816 (0.26)	7858 (2.5)
YANDEX	ydx-malware-shavar	44232 (15.6)	417 (0.1)	11288 (3.9)	88299 (31)
	ydx-adult-shavar	29 (6.6)	1 (0.2)	33 (7.6)	201 (46.3)
	ydx-mobile-only-malware-shavar	19 (0.9)	0 (0)	17 (0.8)	790 (37.5)
	ydx-phish-shavar	58 (0.1)	1568 (4.9)	153 (0.47)	1761 (5.6)
	ydx-mitb-masks-shavar	20 (22.9)	0 (0)	1 (1.1)	9 (10.3)
	ydx-porno-hosts-top-shavar	1682 (1.6)	220 (0.2)	11401 (11.40)	55775 (55.7)
	ydx-sms-fraud-shavar	66 (0.6)	1 (0.01)	22 (0.20)	1028 (9.7)
	ydx-yellow-shavar	43 (20)	1 (0.4)	8 (3.8)	76 (36.4)

7.4.2 Orphan Prefixes

We now look for *orphan prefixes* in GSB and YSB. An entry in the prefix list is called an *orphan* if no 256-bit digest matches it in the corresponding list of full digests. We also look for URLs in the Alexa list which generate an orphan prefix. Table 7.10 presents the results of our findings. Both GOOGLE and YANDEX have orphans.

While GOOGLE has 159 orphan prefixes, for YANDEX the numbers are astonishingly high. 43% for `ydx-adult-shavar`, 99% for `ydx-phish-shavar`, 95% for `ydx-sms-fraud-shavar` and 100% of the prefixes in `ydx-mitb-masks-shavar` and `ydx-yellow-shavar` are orphans. We did not find any URL in the Alexa list that creates an orphan prefix in the GOOGLE's lists. But there are 660 URLs with one parent: the prefix corresponds to one full digest. For YANDEX, we found 271 URLs creating an orphan prefix and 20,220 URLs with one parent.

Table 7.10: Distribution of prefixes as the number of full hashes per prefix. Collision with the Alexa list is also given.

		#full hash per prefix				#Coll. with Alexa list			
	list name	0	1	2	Total	0	1	2	Total
Google	goog-malware-shavar	36	317759	12	317807	0	572	0	572
	googpub-phish-shavar	123	312494	4	312621	0	88	0	88
Yandex	ydx-malware-shavar	4184	279015	12	283211	73	2614	0	2687
	ydx-adult-shavar	184	250	0	434	38	43	0	81
	ydx-mobile-only-malware-shavar	130	1977	0	2107	2	22	0	24
	ydx-phish-shavar	31325	268	0	31593	22	0	0	22
	ydx-mitb-masks-shavar	87	0	0	87	2	0	0	2
	ydx-porno-hosts-top-shavar	240	99750	0	99990	43	17541	0	17584
	ydx-sms-fraud-shavar	10162	447	0	10609	76	3	0	79
	ydx-yellow-shavar	209	0	0	209	15	0	0	15

The presence of orphan prefixes is very difficult to justify. Moreover, the behavior of a browser on these prefixes is not consistent. Some of the orphan prefixes are considered as false positives by Yandex while others are declared as true positives. There are three possible explanations to argue the presence of orphans.

1. First, that there is an inconsistency between the prefix lists and the corresponding lists of full digests. This could be due to a misconfiguration, latency in the update or the result of a development error. This particularly might hold for Google since very few prefixes are orphans.

2. Second, that the services have intentionally noised the database in order to mislead attackers who may try to re-identify URLs from the prefixes.

3. The last argument being that these SB providers might have tampered with their prefixes' database. The presence of a large number of orphans for Yandex proves that it is possible to include any arbitrary prefix in the blacklists.

7.4.3 Presence of Multiple Prefixes

The inclusion of multiple prefixes for a URL is not a hypothetical situation. Instead, our experiments with the databases show that Google and Yandex indeed include multiple prefixes for a URL. We employ the Alexa list and BigBlackList as test vectors for our experiments. The Alexa list has been used in our experiments to determine if Google or Yandex indulge in any abusive use of SB.

In case of BigBlackList, we found 103 URLs creating 2 hits in the Yandex prefix lists. Moreover, we found one URL which creates 3 hits and another one which creates 4 hits. The results on the Alexa list are particularly interesting. We found 26 URLs on 2 domains that create 2 hits each in the malware list of Google. As for the phishing list, we found 1 URL that creates 2 hits. For Yandex, we found 1352 such URLs distributed over 26 domains. 1158 of these URLs create hits in `ydx-malware-shavar` while the remaining 194 are hits in `ydx-porno-hosts-top-shavar`. We present a subset of these URLs in Table 7.11. The large number of such URLs is essentially due to Type I collisions. Nevertheless, these URLs are spread over several domains which shows that Yandex actively includes several prefixes for a URL. This is however less evident for

Table 7.11: A subset of URLs from the Alexa list creating multiple hits in the GOOGLE and YANDEX database.

	URL	matching decomposition	prefix
GOOGLE	http://wps3b.17buddies.net/wp/cs_sub_7-2.pwf	17buddies.net/wp/cs_sub_7-2.pwf	0x18366658
		17buddies.net/wp/	0x77c1098b
	http://www.1001cartes.org/tag/emergency-issues	1001cartes.org/tag/emergency-issues	0xab5140c7
		1001cartes.org/tag/	0xc73e0d7b
YANDEX	http://fr.xhamster.com/user/video	fr.xhamster.com/	0xe4fdd86c
		xhamster.com/	0x3074e021
	http://nl.xhamster.com/user/video	nl.xhamster.com/	0xa95055ff
		xhamster.com/	0x3074e021
	http://m.wickedpictures.com/user/login	m.wickedpictures.com/	0x7ee8c0cc
		wickedpictures.com/	0xa7962038
	http://m.mofos.com/user/login	m.mofos.com/	0x6e961650
		mofos.com/	0x00354501
	http://mobile.teenslovehugecocks.com/user/join	mobile.teenslovehugecocks.com/	0x585667a5
		teenslovehugecocks.com/	0x92824b5c

GOOGLE. We reiterate that the corresponding domains and in some cases even the URLs are re-identifiable. This allows YANDEX, for instance, to learn whether a user prefers the adult site wickedpictures.com or mofos.com through domain re-identification. YANDEX can eventually determine the nationality of a person by the version of xhamster.com he visits. The most interesting example is that of the domain teenslovehugecocks.com. The domain may allow YANDEX to identify pedophilic traits in a user through domain re-identification.

7.5 Mitigations

To the best of our knowledge no prior work has studied SB services from a privacy perspective. Due to the purpose and similarity of the service, our work is strongly related to web-search privacy. Indeed, URLs visited by a user and searched keywords reveal extensive information (see [JKPT07]). Several solutions to improve web-search privacy can be applied to our case and most notably dummy requests (see [GSS+14] for a survey). This solution is currently deployed in Firefox. Each time Firefox makes a query to GSB, some dummy queries are also performed to hide the real one. The dummy requests are deterministically determined with respect to the real request to avoid differential analysis [Ved15]. This countermeasure can improve the level of anonymity for a single prefix hit. However, re-identification is still possible in the case of multiple prefix hits because the probability that two given prefixes are included in the same request as dummies is negligible. Another possible countermeasure consists in redirecting full hash requests through an anonymizing proxy. The essential limitation here is that the client must trust the proxy. Apparently, some proxy services keep server logs of user activity that can be subpoenaed.

Fixing GSB and YSB to prevent any information leakage would ideally require *private information retrieval* [Gol07]. However, none of the existing constructions can scale to the level of SB [SC07, OG12]. Hence, to reduce the amount of information leakage, we propose to query the server *one-prefix-at-a-time*. When a URL has several decompositions creating hits in the prefixes' database, the prefix corresponding to the root node/decomposition is first queried. Meanwhile, the targeted URL is pre-fetched by the browser and crawled

to find if it contains Type I URLs. If the answer from Google or Yandex is positive, a warning message is displayed to the user. Otherwise, if Type I URLs exist, then the browser can query the server for the other prefixes. In this case, Google and Yandex can only recover the domain but not the full URL. In case no Type I URLs exists, a message can be displayed to warn the user that the service may learn the URL he intends to visit.

In order to evaluate this countermeasure, we have developed a proof-of-concept implementation in Python using scrapy [Scr16], a popular web crawler. Among other parameters, scrapy allows to configure the timeout to process DNS queries, using the parameter `DNS_TIMEOUT` and the waiting time to download, using `DOWNLOAD_TIMEOUT`. We set these parameters to 30s and measure the cost incurred for 100 random URLs. The tests were performed on a 64-bit processor laptop computer powered by an `Intel Core i7-4600U CPU` at `2.10GHz` with `4MB cache`, `8GB RAM` and running `Linux 3.13.0-36-generic`. In a sequential setting, fetching and processing of a web page took on an average of 0.17s (for pages which do not cause timeout). We however note that this extra processing incurs no overhead when done in parallel while the client makes a full hash request to the SB server.

We note that the crawler is configured to follow a restricted crawling strategy: it can only recover links on the target web page. As a consequence, the crawler may fail to find a Type I URL even when it actually exists. In our experiments, this strategy found Type I URLs in 90% of the cases. A thorough albeit costlier approach would consist in crawling the SLD of the target URL. This ensures that the crawler never errs. The same experiment with a complete SLD crawl required roughly 3 times more time, *i.e.*, 5s. The underlying crawling strategy hence presents a trade-off between privacy and robustness achieved by the countermeasure.

7.6 Summary

Safe Browsing services are valuable tools to fight malware, phishing and other online frauds. Unlike other Safe Browsing vendors, Google and Yandex have made sincere efforts to render their services as private as possible. However, the effect of their anonymization efforts has been largely unclear. We have quantified the privacy provided by these services and have shown that the underlying anonymization technique of hashing and truncation fails when the server receives multiple prefixes for certain classes of URLs.

Our observations on the Yandex database and to a lesser extent on that of Google show that it is possible to tamper these databases. These instances could either be deliberate attempts or the results of development errors/misconfigurations or latency. Irrespective of the underlying cause, the service readily transforms into an invisible tracker that is embedded in several software solutions.

As a part of responsible disclosure, we contacted the GSB team. Unfortunately, we cannot reveal the communication due to a non-disclosure agreement. We also contacted the Mozilla Firefox security team and here is an excerpt of the feedback:

> "*We have long assumed (without the math to back it up) that if Google were*

evil it could seed the list with prefixes that allowed it to detect whether a few users visited a few select targets."

Chapter 8

(In)Security of Safe Browsing

Contents

8.1 Introduction **145**
8.2 Related Work **146**
8.3 Attacks on Safe Browsing **147**
8.3.1 Threat Model 147
8.3.2 Attack Routine 148
8.3.3 False Positive Flooding Attacks 149
8.3.4 "Boomerang" Attacks 150
8.3.5 Impact 151
8.4 Feasibility of Our Attacks **152**
8.4.1 Generating Domain Names and URLs 152
8.4.2 Generating Deadly Domains 153
8.4.3 Comparing the Domain Topologies 155
8.5 Countermeasures **156**
8.5.1 Lengthening the Prefixes 156
8.5.2 Probabilistic Solutions 157
8.6 Ethical Considerations on Attack Demonstration **158**
8.7 Responsible Disclosure and Impact **159**
8.8 Summary **162**

8.1 Introduction

Google and Yandex Safe Browsing are high impact security infrastructures. While, these SB tools aim to provide the much needed security to online users, they should also be secure in their own right as any vulnerability in the system has the potential to affect billions of users. The goal of this chapter is to provide an assessment of the security of GSB. In this chapter, we identify vulnerabilities in GSB and present several easy ways to mount attacks. Conceptually, the main goal of the attacks is to increase the traffic towards GSB servers and its clients in the form of a DoS.

The attacks described here leverage the data representation employed on the client's side. The current representation stores 32-bit prefixes of malicious URLs, which generates false positives. We study two classes of attacks, namely, *false positive flooding attacks* and *boomerang attacks*.

The *false positive flooding attacks* aim to increase the load on GSB servers by increasing the false positive probability exhibited by the local database. With the increased number of false positives, the adversary forces clients to contact the servers even for benign URLs. Typically, every time a user visits a benign URL, he is forced to send a request to the server to resolve the ambiguity. If several popular web pages are targeted, then the SB service can be brought to its knees.

Our *boomerang attacks* on the other hand, target the traffic from GSB servers to its clients. The adversary creates many malicious web pages sharing the same 32-bit prefix digest. After being discovered by Google, they will be included in GSB servers and their prefix included in the local database of all the clients. Each time, a client makes a request for this particular prefix, he receives the full digests of all the URLs created by the adversary.

In order to mount the attacks, an adversary needs to forge URLs with malicious content corresponding to certain 32-bit prefixes. This amounts to computing pre-images or second pre-images for 32-bit digests. Clearly, this is highly feasible and in fact very fast to obtain.

It should not come as a surprise that the attacks simply exploit the fact that digests of small lengths are easily invertible. In other words, smaller prefixes are the real source of vulnerability in the SB architecture. Also, the first immediate approach to patch the vulnerability would be to increase the prefix length. With a longer prefix, pre-image and second pre-image attacks can easily be thwarted. Furthermore, as described in Chapter 6, the Safe Browsing API requires that clients should be able to handle any prefix size between 32 bits to 256 bits, hence this patch provides a countermeasure that is compatible with the existing protocol requirements. Unfortunately, with the lessons learned from Chapter 7, this countermeasure is not as promising as it sounds since it would expose users to re-identification and eventual tracking. Towards the end of this chapter, we discuss this countermeasure along with other probabilistic solutions and some of the underlying issues.

This chapter is organized as follows. We first present the related work in Section 8.2. Our general attack routine, the false positive attacks and the boomerang attacks are described in Section 8.3. In Section 8.4, we discuss the feasibility of our attacks. Motivated by our results, we discuss in Section 8.5 the different countermeasures to prevent our attacks, namely, randomization and prefix lengthening.

8.2 Related Work

No prior work has studied the security of GSB and YSB. However, several other web-malware detection systems, such as *Virtual Machine client honeypots*, *Browser Emulator client honeypots*, *Classification based on domain reputation* and *Anti-Virus engines* have been extensively studied in the past. The VM-based systems [WBJ+06, MBGL06, MBD+07] typically detect exploitation of web browsers by monitoring changes to the OS,

such as the creation of new processes, or modification of the file system or registries. A browser emulator such as [CKV10, Naz09] creates a browser-type environment and uses dynamic analysis to extract features from web pages that indicate malicious behavior. In the absence of malicious payloads, it is also possible to take a content-agnostic approach to classify web pages based on the reputation of the hosting infrastructure. Some of these techniques [APD+10, FKP10] leverage DNS properties to predict new malicious domains based on an initial seed. Finally, Anti-Virus systems operate by scanning payloads for known indicators of maliciousness. These indicators are identified by signatures, which must be continuously updated to identify new threats.

A GSB like blacklisting mechanism is also adopted by Dropbox[1] which uses hashing to know when a user attempts to share a copyrighted file. Dropbox maintains a big blacklist of hashes known to be those corresponding to files they can not legally allow to be shared. When a user shares a link to a file, it checks the file's hash against the blacklist. A match in the blacklist implies that the file cannot be shared.

Our DoS attacks retain some flavor of *algorithmic complexity attacks*, where the goal is to force an algorithm to run in the worst-case instead in the average-case. Our security analysis of SB distinguishes itself from previous works on algorithmic complexity attacks in two aspects. First, the verification algorithm that we attack is distributed over the client and the server. While in the previous works, the target algorithm was run solely on the server side. Second, the data representations used by these SB services do not have different worst-case and average case complexities. Instead, these data representations entail false positives. Increasing the false positive probability implies increasing the number of requests towards the SB servers made by honest users: Safe Browsing will be unavailable.

8.3 Attacks on Safe Browsing

In this section, we present our attacks on GSB. Since, YSB employs an identical architecture, we later conclude that our attacks trivially extend to the service proposed by Yandex. In the following, we first describe our threat model and then in the sequel, we develop our attacks. For the attacks, we first discuss the attack routine, *i.e.*, the steps followed by the adversary and then we develop them.

8.3.1 Threat Model

There are three possible adversarial goals: increase the traffic towards SB servers, or increase the traffic towards the clients or both. A considerable increase in the traffic towards the server leads to DoS: the SB service becomes unavailable for honest users. Symmetrically, an increase in the traffic towards a client could quickly consume its bandwidth quota.

An adversary against GSB has the following capabilities. She can create and publish malware and phishing websites with URLs of her choice and can submit them to Google.

[1] https://techcrunch.com/2014/03/30/how-dropbox-knows-when-youre-sharing-copyrighted-stuff-without-actually-looking-at-your-stuff/

This is equivalent of being capable of inserting prefixes in the local database of each client.

8.3.2 Attack Routine

Our attacks follow a three-phase procedure (see Figure 8.1): forging malicious URLs, including these URLs in GSB, and updating the local database. In the following, we discuss these phases in detail.

1. **Forging malicious URLs:** The first step of the attack aims to generate malicious URLs corresponding to prefixes that are not currently included in GSB. This requires an adversary to find unregistered domain names and URL paths on which malicious content could be later uploaded. These canonical URLs are so chosen such that their digests yield the desired prefixes. Hence, in general our attacks are either pre-image or second pre-image attacks on the underlying hash function but restricted to truncated 32-bit digests. We defer the discussion on the feasibility of generating these URLs to Section 8.4. We note that rendering these URLs malicious simply requires to upload malicious content on it. This constitutes Step 1 in Figure 8.1.

2. **Including URLs in GSB:** After forging malicious URLs and adding malicious content, the second step of our attack consists in including the newly forged URLs in GSB. There are three ways to do so. Either these pages get detected by the Google crawlers themselves, or one explicitly asks Google to crawl it[2] which eventually would detect it as malicious, or one may also use the Safe Browsing Reporting tool to report these URLs as malicious using the following interfaces:

   Malware: google.com/safebrowsing/report_badware/

   Phishing: google.com/safebrowsing/report_phish/

   Finally, in order to increase the visibility of these URLs, an attacker may also submit these URLs to Google's sources such as phishtank.com and stopbadware.org. We note this is the only step that the adversary cannot control and hence she has to wait until Google flags the newly found URL as malicious. This is a sequential process as depicted through Step 2-4 in Figure 8.1.

3. **Local database update:** Once these malicious URLs reach the GSB server, it diffuses them by sending an updated local database to clients in order to incorporate them. In this way, the adversary has established a data flow with all the end users through Google. This update process is shown in Step 5-7 in Figure 8.1.

At the end of the afore-described phases, there are several new malicious links on the web and the local database copy of each user has been accordingly updated. We highlight that the vulnerability in GSB comes from the fact that the local data structure has a false positive probability. With the current prefix size of 32-bits, an adversary can increase the false positive probability by inserting certain prefixes corresponding to malicious URLs.

[2] https://support.google.com/webmasters/answer/1352276?hl=en

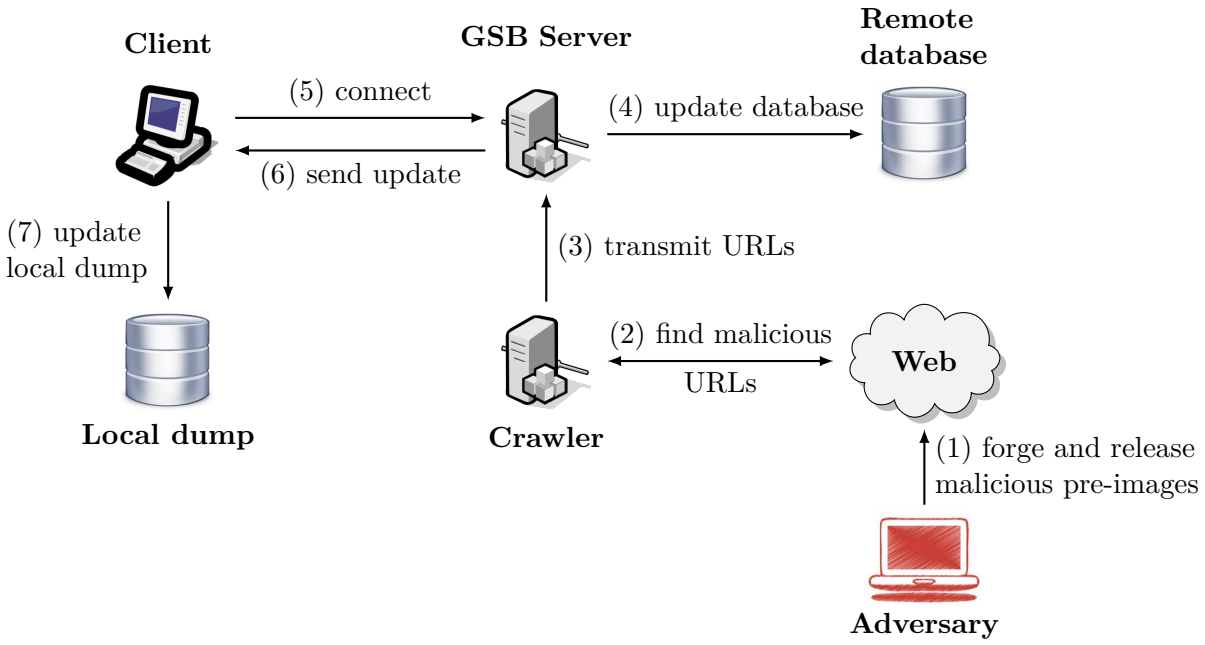

Figure 8.1: The attack routine: An adversary forges malicious URLs and diffuses them. These URLs then get detected and indexed by the GOOGLE crawlers and eventually reach the clients through the database update.

In the remaining part of this section, we explain how to judiciously create URLs in order to pollute GSB.

8.3.3 False Positive Flooding Attacks

An adversary using pre-images or second pre-images may force the GSB client to send queries to the distant server more frequently than required. The actual attack depends on whether pre-images or second pre-images are found in the first phase of the attack routine. We describe below these two attacks.

8.3.3.1 Pre-image Attacks

Let us consider a prefix which is not currently in the local database. Then, our pre-image attacks essentially rely on malicious pre-images of this prefix, *i.e.*, malicious URLs that generate the same prefix. Pre-image attacks on GSB consists in generating pre-images in the first phase of the attack routine. In case of a delta-coded table, this results in an increased false positive probability. This is because at the end of the third phase, the table represents a much larger set of prefixes. In fact, the false positive probability of a delta-coded table is $\frac{\#\text{prefixes}}{2^{32}}$. Hence, the false positive probability linearly increases with the number of new prefixes added to the local database at the end of the third phase of the attack routine.

As a result of the increased false positive probability, many URLs will create a hit during the local verification. Eventually, the traffic towards the SB server would be increased, since the server would be contacted more frequently than necessary for a confirmation of a true positive. In other words, this challenges the efficiency of the service.

8.3.3.2 Second Pre-image Attacks

This attack consists in generating second pre-images in the first phase of the attack routine. While this attack is in principle the same as the afore-described pre-image attack, the impact of second pre-image attacks can however be much more severe. In order to exemplify this, let us consider a non-malicious pre-existing URL, say: http://www.conf.org/2016/ and its 32-bit prefix `0x07fe319d`, an adversary would exhaustively search for a second pre-image of the 32-bit prefix in the first phase of the attack routine. An illustrative example is http://www.malicious-conf.org/2016/115124774. Such a URL (with malicious content) is then released on the Internet in the second phase. Google crawlers eventually detect the URL as malicious and add it to its database. The prefix dump on the client side is accordingly updated to incorporate the newly found URL. Now, whenever a user visits the actual conference web page, the corresponding prefix creates a hit in the local database. Consequently, the browser is forced to request the server to get full digests for the concerned prefix. The threat towards the servers is further exacerbated when an adversary can insert prefixes corresponding to popular web pages. Since these web pages are frequently visited, the probability that a client creates a hit in the local database is high. Consequently, the number of queries can grow quickly and would consume the network bandwidth.

8.3.4 "Boomerang" Attacks

An adversary may further magnify the amount of bandwidth she can target at SB clients by mounting what we refer to as "*boomerang*" attacks. The term "boomerang" attack is borrowed from [Sch05].

Boomerang attacks come into effect when a client sends a small packet that elicits a full hash for a certain prefix. In reply, a large response packet is returned to the client by the SB server. We note that a few such requests can significantly contribute in the consumption of the allowed bandwidth to a client.

In order to mount boomerang attacks, the adversary generates t (second) pre-images (in the form of URLs) of a target prefix in the first phase of the attack routine. At the end of the third phase, the SB server includes the common prefix in the clients' local database. Now, whenever a client accesses one of these URLs, the corresponding prefix creates a hit in the local database, and hence the client sends the prefix to the server eliciting the corresponding full hashes. In reply, the client receives at least t full hashes, symmetrically forcing the GSB servers to send this data. Consequently, network bandwidth on both sides is increased. We highlight that boomerang attacks are particularly interesting in the case where prefixes corresponding to popular web pages are targeted. Since these pages are frequently visited, a request for the full hashes would quickly consume the allowed bandwidth to a client. Furthermore, since the client has to store these full hashes until an update discards them, these full hashes also consume the browser's cache.

8.3.5 Impact

8.3.5.1 Measuring the Increase in Traffic

We have seen that in case of boomerang attacks, the increase in traffic towards clients can be measured by the number of pre-images found. After a week of computation (see Section 8.4), we obtained an average of 100 second pre-images per prefix. These prefixes correspond to popular web pages.

In order to compare the increase in traffic, we first present in Table 8.1 the statistics on the number of prefixes in the phishing and malware lists, and the number of full digests matching them. This gives the baseline for comparison. In the worst case, the response contains two SHA-256 digests. Considering that GOOGLE servers currently send at most 2 full digests per prefix, it is possible to achieve an amplification factor of 50 in the average case (since on an average we found 100 second pre-images per prefix). The maximum number of second pre-images that we found at the end of one week was 165. This leads to an amplification factor of 82 in the best case for an adversary.

Table 8.1: Distribution of the reply size in the number of full hashes per prefix for malware and phishing blacklists.

# full hashes/prefix	Malwares	Phishing
0	36	123
1	317759	312494
2	12	4

In order to precisely determine the size of the request and response, we have used Mozilla Firefox as the client together with OWASP ZAP proxy[3] and Plug-n-Hack, a Firefox plugin. Firefox with the plugin enabled allows the proxy to intercept each communication between the client and the SB server. As measured, the actual request size is 21 Bytes, while a response with at least 1 full digest has a size of 315 Bytes. Hence, if an adversary is able to generate 100 (second) pre-images for a prefix, then the server would send a response of size 30 KB. The size of the response received by the client linearly increases with the number of pre-images found. With only 10^4 pre-images the adversary can force the server to send 3 MB of data to the client. For the sake of comparison, the local database updates sent to a client during a period of 24 hours was measured to be around 400 KB.

8.3.5.2 Impact on Other Services

It is worth noticing that our attacks on GSB have a wide reach. In the first place, our proposed attacks directly affect the usability of the browsers. All popular browsers including Chrome, Chromium, Firefox, Safari and Opera include GSB as a feature. Even worse, as discussed in Chapter 6, several big companies such as Facebook and Twitter import GSB in their social networking services. These services have billions of users and the user activity on these websites is extremely high.

[3] https://github.com/zaproxy/zap-core-help/wiki

The impact of our attacks is very wide because the client code is available as open-source, hence any external service can use it together with the Safe Browsing API to build its own SB solution. This makes the new service also vulnerable. The most notable example of such an external service is YSB. Since the client side implementation of GSB and YSB is identical, all our attacks on GSB trivially extend to YSB.

8.4 Feasibility of Our Attacks

In this section we empirically study the feasibility of our attacks. More precisely, we consider the feasibility of generating polluting URLs. We highlight that the URL generation is the only attack step with a considerable overhead. URL generation requires finding (second) pre-images corresponding to certain prefixes.

It is believed that for a cryptographic hash function h producing ℓ-bit digests, the basic complexities for finding pre-images and second pre-images is 2^{ℓ}. Hence, if ℓ is large, (second) pre-image attacks are not feasible. However, in case of GSB, the adversary exploits the fact that the SHA-256 digests of malicious URLs are truncated to 32-bits. Truncation allows the adversary to obtain pre-images and second pre-images in reasonable time, *i.e.*, only 2^{32} brute-force operations. We hence employ a brute-force search in our attacks. Since, finding a pre-image or a second pre-image through brute-force entails the same cost, we do not distinguish them in the sequel unless it is required.

8.4.1 Generating Domain Names and URLs

We note that a pre-image for a 32-bit prefix can be computed using brute-force search in a few hours on any desktop machine. Relying on this fact, we have written an attack-specific pre-image search engine in Python implementing a brute-force search. It was built with a specific goal of searching multiple second pre-images for 1 million popular web pages in the Alexa list. The ensuing results therefore determine the feasibility of mounting a second pre-image based attack on a target benign URL in the Alexa list. Moreover, the engine also allows to generate pre-images required for a pre-image based false positive flooding.

Since a brute-force search is highly parallelizable, we exploit the module Parallel Python[4]. In fact, two levels of parallelization can be achieved. At a higher level, the search for multiple pre-images can be parallelized. Consequently, two pre-images can be obtained in roughly the same time as the search for one pre-image. At a lower level, the generation of a single pre-image can also be parallelized by dividing the interval of search, $[0, 2^{32})$ into sub-intervals.

We also check the availability of the domain name corresponding to each URL. This is necessary since the adversary should own the domain name to be able to upload malicious content on it. This verification is performed using the python module `pywhois`: a wrapper for the Linux `whois` command. Finally, to ensure that the URLs are human readable, we have employed the Fake-factory[5] (version 0.2), a python package to generate fake but

[4] http://www.parallelpython.com/
[5] https://pypi.python.org/pypi/fake-factory/0.2

human readable URLs.

All our experiments were performed on a cluster with CPython 2.6.6 interpreter. The cluster runs a 64-bit processor powered by an Intel QEMU Virtual CPU (cpu64-rhel6), with 32 cores running at 2199 MHz. The machine has 4 MB cache and hosts Centos Linux 2.6.32. Figure 8.2 presents the results obtained by our search engine at the end

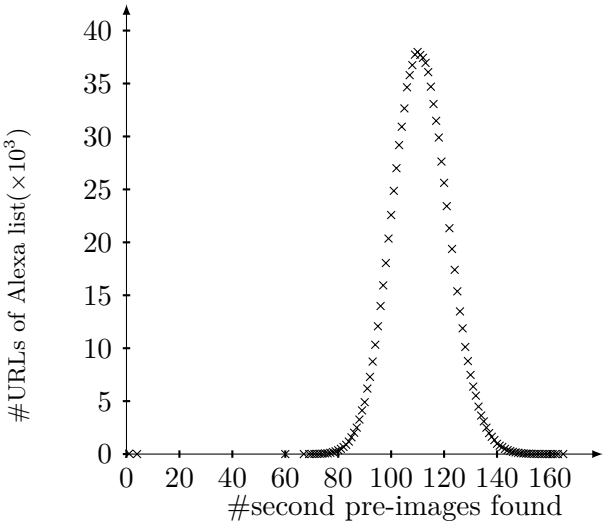

Figure 8.2: Number of second pre-images found per URL in the Alexa list of 1M URLs.

of 1 week. The total number of second pre-images found was 111,027,928, *i.e.*, over 111 million. Since the Alexa list contains 1 million entries, as expected the number of second pre-images found per URL is highly concentrated around 100. For around 38,000 URLs in the Alexa list, we found 110 second pre-images per URL. The tail corresponds to the URLs for which we found the largest number of second pre-images. A summary of the results for the tail is presented in Table 8.2.

Table 8.2: Prefixes and the domains in the Alexa list for which most second pre-images were found. A sample second pre-image is also provided. The search gave 160 second pre-images for 2 websites.

Prefix	# Second pre-images	Alexa Site	Sample second pre-image
0xd8b4483f	165	http://getontheweb.com/	http://9064606pearliefeil.com/
0xbbb9a6be	163	http://exqifm.be/	http://ransomlemke.com/id15926896
0x0f0eb30e	162	http://rustysoffroad.com/	http://62574314ginalittle.org/
0x13041709	161	http://meetingsfocus.com/	http://chloekub.biz/id9352871
0xff42c50e	160	http://js118114.com/	http://easteremmerich.com/id12229774
0xd932f4c1	160	http://cavenergie.nl/	http://41551460janaewolff.com/

8.4.2 Generating Deadly Domains

The afore-presented results show that the generation of multiple pre-images of a 32-bit digest is time-efficient. However, for an attack to be successful, it is also extremely important that the attack be cost efficient. As described earlier, an adversary upon finding an available pre-image has to purchase the domain name. The cost of each is typically $6-10 for a `.com` top-level domain. As boomerang attacks require an adversary to generate multiple (second) pre-images, the final cost of an attack might become prohibitive.

A more cost-efficient solution is to purchase one single domain to cover several prefixes. Let us say that the chosen domain is `deadly-domain.com`. The adversary then simply requires to create several malicious and non-malicious links on the domain. To this end, two strategies can be employed. We describe below these strategies. In order to exemplify these, let us consider for instance that the adversary wishes to use `deadly-domain.com` to cover three prefixes: `prefix1`, `prefix2` and `prefix3`.

1. **Same Depth Strategy**: Search for malicious tags at the same depth, such as `maltag1`, `maltag2` and `maltag3` such that:

   prefix(SHA-256(`deadly-domain.com/maltag1`)) = `prefix1`,
   prefix(SHA-256(`deadly-domain.com/maltag2`)) = `prefix2`,
   prefix(SHA-256(`deadly-domain.com/maltag3`)) = `prefix3`.

   These tags may correspond to name of files, such as `.html`, `.php`, *etc.* Once these tags are found, these pages or files are then linked to malicious content. Table 8.3 presents sample second pre-images for popular web pages generated using the same depth strategy.

Table 8.3: Same depth strategy: Sample second pre-images for popular web pages.

malicious URL	popular domain	prefix
deadly-domain.com/4294269150	google.com	0xd4c9d902
deadly-domain.com/21398036320	facebook.com	0x31193328
deadly-domain.com/5211346150	youtube.com	0x4dc3a769

2. **Increasing Depth Strategy**: Search for malicious tags at increasing depth, such as `maltag1`, `maltag2` and `maltag3` such that:

   prefix(SHA-256(`deadly-domain.com/maltag1`)) = `prefix1`,
   prefix(SHA-256(`deadly-domain.com/maltag1/maltag2`)) = `prefix2`,
   prefix(SHA-256(`deadly-domain.com/maltag1/maltag2/maltag3`)) = `prefix3`.

   These tags correspond to the name of directories. Once these tags are found, malicious files are uploaded in these directories. Table 8.4 presents sample second pre-images for popular web pages generated using the increasing depth strategy.

Table 8.4: Increasing depth strategy: Sample second pre-images for popular web pages.

malicious URL	popular domain	prefix
deadly-domain.com/4294269150/	google.com	0xd4c9d902
deadly-domain.com/4294269150/3263653134/	facebook.com	0x31193328
deadly-domain.com/4294269150/3263653134/2329141652/	youtube.com	0x4dc3a769

The malicious tags are randomly generated using the previous search engine. Once all the tags are found, the malicious URLs found can be released on the web. If GSB considers all these URLs as malicious, then it will include all the three prefixes in their blacklists. Hence, only one domain suffices to cover three prefixes. The same strategy can

be used for second pre-images based attacks, where popular websites are targeted. In this case, the prefixes correspond to the domains in the Alexa list. Last but not least, this also allows to generate multiple pre-images: it suffices to fix a prefix and search for several malicious tags.

However, there is a small caveat in the above strategies. Considering that several URLs on the same domain host malicious content, the GSB servers may decide to blacklist the entire domain by adding only the digest prefix of the domain name in their lists. This results in the degenerate case, where only one prefix gets included in the blacklist. In order to circumvent this issue, the adversary would need to create intermediary non-malicious pages on the domain which correspond to useful or safe to browse content. Without these pages, there is a chance that only the initial domain name is flagged as malicious.

8.4.3 Comparing the Domain Topologies

As described in the previous sections, three strategies are possible for generating a malicious domain and the corresponding URLs: the naive strategy of generating one domain name per prefix, the same depth strategy and the increasing depth strategy. The final topologies of the domains obtained through these are schematically presented in Figure 8.3.

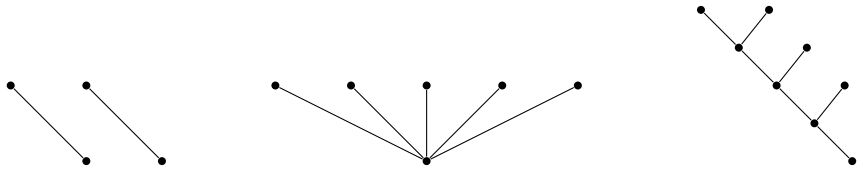

Figure 8.3: Three possible topologies of the domains generated by an adversary are depicted. The lowest node in the graphs represent the root domain, while the other nodes correspond to the files or the directories created on the root domains. The topology (a) represents the case where a new domain is created for each prefix, (b) represents the domains generated by the same depth strategy, and (c) represents those corresponding to the increasing depth strategy.

We reiterate that while the naive strategy is highly parallelizable, it may be cost prohibitive. The same depth strategy however assures the same level of parallelization while remaining cost efficient. Compared to these strategies, the increasing depth strategy is relatively less parallelizable since the malicious tag generation is sequential. Indeed, the adversary has to wait for the first malicious tag to be able to generate the next one. While search for multiple pre-images cannot be parallelized, yet the search for a single pre-image is parallelizable.

Despite its disadvantages, the increasing depth strategy greatly impacts the size of the full hash request and that of the corresponding response. In order to understand this, let us suppose that all the corresponding malicious URLs get included in the blacklists. If a client visits the final URL, he first decomposes the URL into smaller URLs and checks if their corresponding prefixes are in the local database. Since, these decompositions correspond exactly to the URLs generated by the attacker, all these prefixes create hit in the local database. Consequently all the prefixes are sent to the server. The exact number

Table 8.5: Comparative summary of the different strategies for generating malicious domains. The comparison is relative to a scenario without any attack on the service. The symbol '∅' denotes that no change could be obtained by the strategy. We use the symbol '+' to indicate the relative magnitude of the different values. Adversary's complexity inversely relates to the level of parallelization that she can achieve.

	Client's request		Server's response		
Strategy	size	frequency	size	frequency	Adversary's complexity
Naive	∅	+	+	+	+
Same depth	∅	+	+	+	+
Increasing depth	++	+	++	+	++

of prefixes sent to the server is equal to the depth of the tree (as in Figure 8.3). In case only the URLs up to a depth d are blacklisted by the SB service, then the number of prefixes sent is exactly d. Symmetrically, the server in response has to send back all the full hashes corresponding to all the received prefixes. In this sense, the final URL is the deadliest URL. We conclude this section by summarizing the comparison between these strategies in Table 8.5.

8.5 Countermeasures

The design of a good countermeasure to our DoS attacks must be easy to deploy, ensure compatibility with the current API and entail acceptable cost in terms of time and memory at the client and the server side. We investigate two countermeasures to our attacks. The first and the obvious one is to increase the size of the prefix. The second solution consists in randomizing the system with keys.

8.5.1 Lengthening the Prefixes

The core of our attacks is the computation of pre-images and second pre-images. As empirically shown in the previous section, 32-bit prefixes are not enough to prevent those attacks and thus increasing their length is the obvious choice. Unfortunately, it has some effect on the size of the data structure at the client side. Furthermore, the designers of Safe Browsing want to keep this size below 2MB. From Table 6.2 (See Chapter 6 §6.3.3), we observe that delta-coded tables do not scale well if we increase the prefix length. For 64-bit prefixes, the size of the data structure gets tripled. The size of the Bloom filter remains immune to the change in the prefix size, but unlike the delta-coded table, the data structure would no longer be dynamic which is critical in the context of Safe Browsing. It can be argued that storing the 20.3MB of all the full digests at the browser side is not a big issue. Unfortunately, publishing the full digests has very dangerous side effects. Indeed, the URLs included in GSB may correspond to legitimate websites that have been hacked. With the full digests, hackers can use GSB to identify weak websites and increase the problem for the corresponding administrators. Recovering URLs from full cryptographic digests was demonstrated to be possible when the MD5 digests of the URLs censored in Germany were published and inverted (see `bpjmleak.neocities.org`).

More generally, an intermediate prefix size between 32 bits (current size) and 256 bits (full digest), for instance prefixes of 80 bits may appear to solve the problem. Let us focus on 80 bits, since this ensures the minimum acceptable security level in cryptography to prevent pre-image attacks. However, the privacy analysis of Google and Yandex Safe Browsing (*Cf.* Chapter 7) shows that with the current size of the web, 64-bit prefixes will in fact increase the privacy risks (refer to Table 7.3 from §7.2). Indeed, the results suggest that, there were around 60×10^{12} distinct URLs on the web in 2013. If these URLs are used to generate 64-bit prefixes, then each prefix can correspond to a maximum of 2 URLs. For domain names, the situation is even worse. There were around 271×10^6 distinct domain names in 2013, which yields 1 domain name per prefix. The situation becomes dramatically worse with a prefix size of 80 bits. Relying on these results, we argue that lengthening the prefix size is not a viable solution to defend against our proposed attacks unless users' privacy can be put at stake.

8.5.2 Probabilistic Solutions

Since our attacks retain some flavor of algorithmic complexity attacks, we explore existing solutions to prevent such attacks. The first countermeasure proposed to defeat algorithmic complexity attacks was to use universal hash functions [CW77] or message authentication codes (MAC) [MVO96]. Such methods work well for problems in which the targeted data structure is on the server side. The server chooses a universal hash function or a key for the MAC and uses it to insert/check elements submitted by clients. The function or the key is chosen from a large set so that it is computationally infeasible for an adversary to either guess the function/key or pre-compute the URLs for all the possible values. We further highlight that all the operations made on the data structure can only be computed by the trusted server.

For GSB, the situation is different. The data are provided by Google and inserted in the data structure by the client. It implies that all the prefixes and any keys can not be kept secret to the client and are therefore known by an adversary. With the knowledge of 32-bit prefixes, the adversary can mount false-positive flooding attacks or boomerang attacks.

The first solution is to use a MAC directly on the URL prefixes. A key is chosen by Google for each client and then shared. The prefixes received are unique to each client. An adversary can mount a second pre-image attack on a given user if she knows his key. But it can not be extended to other users without the knowledge of their key. This solution defeats both pre-image and second pre-image based attacks. However, it requires that Google recomputes the prefixes for each client. We propose this countermeasure to protect important and big clients such as Facebook or Twitter. However, extending this to all users entails scalability issues.

Another solution could be that all the clients share the same function/key, but this key is renewed by Google every week or month. The key is used to compute the prefixes as in the previous solution. An adversary can pollute Google Safe Browsing only for a short period of time as he also knows the key. In this strategy, all the prefixes must be

re-computed on a regular basis by GOOGLE servers with a different function or key and diffused to all the clients. However, this may represent a high communication cost once per week/month.

8.6 Ethical Considerations on Attack Demonstration

In order to test whether our attacks are viable, we attempted to mount a second pre-image attack. The domain in question was a conference web page. To this end, we created a phishing web page on a free web hosting server. We also created several links on the page that redirected visitors to domains considered malicious by GOOGLE. The web page was reported to GOOGLE and other harvesting platforms such as PHISHTANK. A URL reported to PHISHTANK is considered malicious when it gets sufficient number of votes from volunteers evaluating the associated threat. We created fake accounts to have enough sybil volunteers. Five votes from our sybil accounts sufficed to flag the web page as malicious.

However, the web page was not detected as malicious by GOOGLE and consequently the prefix did not get included in the local database. Multiple explanations are possible: 1) Web pages created on free hosting servers are often ignored by GOOGLE during the crawling process. Indeed, a search in the GOOGLE search engine for several important keywords to identify the malicious web page did not return any result even after one month of the website launch. 2) GOOGLE may have analyzed the web page but did not consider it to be malicious. This was verified using the diagnostic tool which revealed that the web page was indeed analyzed by GOOGLE but was found to be safe. 3) This explanation somewhat extends the previous one. GOOGLE did not consider the web page to be malicious because it had no history of affecting any end user.

To conclude, a demonstration of our attack could have succeeded if the web page had been hosted on a registered domain and had affected some set of users. To the best of our understanding, this would have required us to behave exactly like a malicious entity. Furthermore, in order to have a considerable impact on users, instead of attacking a conference web page, we should have created a phishing banking web page or a web page for a popular social networking site. Eventually, the web page should have been spread through spam emails to increase the visibility and the attack surface. Indeed, this would have helped to better establish the associated threat. However, due to ethical considerations we did not mount such a full fledged attack. Despite the limited nature of our attack demonstration, Kaspersky[6] (a popular threat analysis tool) considered our web page as a phishing threat (see Figure 8.4). We note that the web page was never reported to Kaspersky directly and hence Kaspersky through its own means discovered the web page and established the associated threat. This result gives some promising evidence that a motivated attacker may indeed succeed to reconstruct our attack against GSB.

[6] www.kaspersky.com

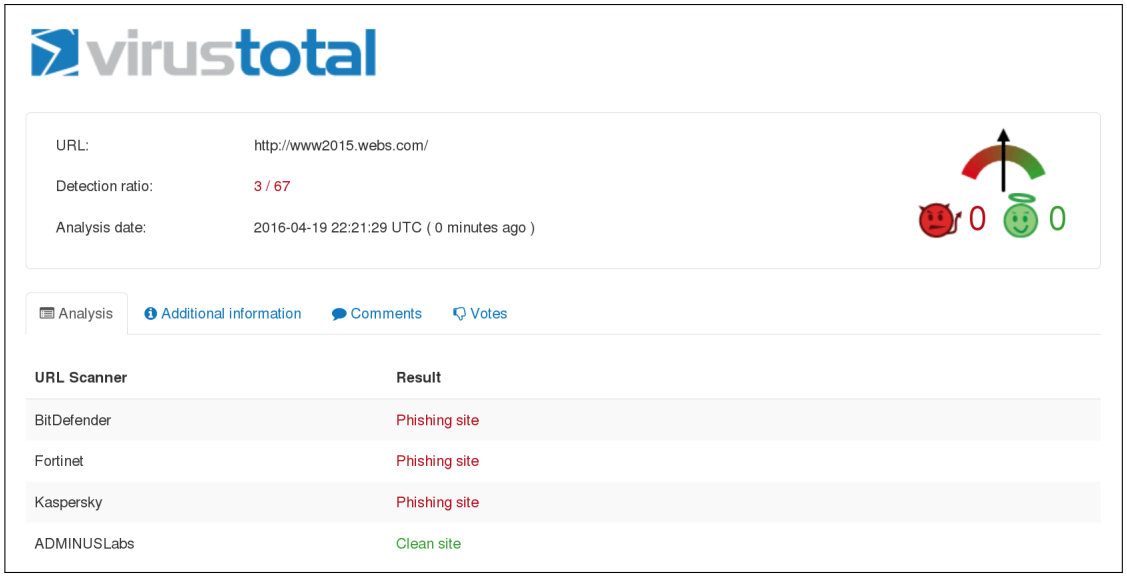

Figure 8.4: An excerpt of the analysis by VirusTotal (http://virustotal.com/) of our phishing page.

8.7 Responsible Disclosure and Impact

We contacted Mozilla Firefox, Google and Yandex as a part of responsible disclosure. We did not get any reply from Yandex and Mozilla. However, a search on Mozilla's bug reporting and discussion portal allowed us to learn about Mozilla's take on these attacks.[7] Apparently, some of the attacks are known to Firefox developers.

> "*It is computationally very feasible to collide the *original* 32-bit SHA-256 prefixes with a small botnet. (e.g.* `g00gle7074559436.org/` *collides* `google.com/`*;* `http://www.sjeng.org/4806321590.html` *collides known malware).*"
>
> "*... Google in fact does not care about potential floods to the completion server, and has enough (dynamic) resources to deal with it.*"
>
> "*Turning this into a real attack would require some sort of agreement among everyone posting malware/phishing sites to only do so at colliding URLs, and the result would then be a sort of DDoS against the relevant Google servers, or slowed loading of another target site.*"

While, false positive flooding attacks towards SB servers appear to be known, the developers are oblivious to our boomerang attacks that target SB users. Moreover, a possible countermeasure is also discussed, which is similar to the probabilistic solutions discussed earlier in the previous section.

> "*So you generate a random number R and for all prefixes, store* `sha32(prefix|R)`*. Then when you visit a page, you check whether* `sha32(sha32(url)|R)` *is in the database?*"

[7] https://bugzilla.mozilla.org/show_bug.cgi?id=782106

Unlike Mozilla and YANDEX, GOOGLE however did respond to one of our emails, but only to inform us that prefix lengthening would counter all the proposed attacks. Here is an excerpt of the email:

> "*We are aware of the potential issues that could arise from blacklisting a popular hash prefix. We believe we have the necessary protections in place to avoid the DoS scenario that you're describing. Note that the current API already supports lengthening of hash prefixes. If a high traffic prefix is being added to the Safe Browsing list we have the ability to not add it to the list or to lengthen it to 32 bytes. Lengthening problematic prefixes addresses the issues that you're describing in the paper at the cost of increasing the update bandwidth. We're currently in the process of building the next version of the Safe Browsing API which will have the ability to serve prefixes of arbitrary length (instead of just 4 bytes and 32 bytes).*"

However, when we followed up to discuss the impact of prefix lengthening on re-identification attacks and privacy, we did not receive a follow up reply.

Apparently, a year later on April 21, 2016, The Washington Post[8] reported that google.com was being blacklisted by the Safe Browsing Diagnostic tool. Figure 8.5 shows a screenshot of the tool on that day, which considers that google.com probably hosts malwares. Numerous other popular websites were also flagged as potentially malicious. These include github.com and tumblr.com among others. Strangely, GOOGLE declined to comment on the issue. While the exact reason seems to be unknown, it is possible that the reason behind this incident is that intelligently scripted URLs targeting popular domains got included into GOOGLE's blacklists.

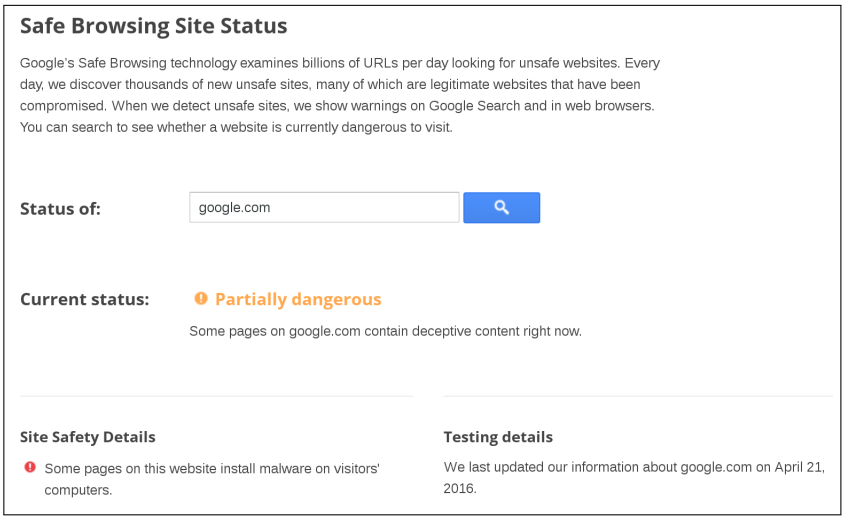

Figure 8.5: The Safe Browsing diagnostic tool shows google.com as potentially malicious. Accessed on April 21, 2016.

In June 2016, GOOGLE released the Safe Browsing API version 4 and declared the

[8] http://www.washingtonpost.com/news/the-switch/wp/2016/04/19/why-google-is-warning-that-google-com-is-partially-dangerous/

earlier version 3 to be deprecated.[9] Google adverts that support for existing (v3) clients will be provided through early 2017. Though not formally acknowledged, some of the changes made in v4 take into account our attacks. The end result is that the security of GSB is indeed improved but no changes have been made to increase users' privacy.

The Safe Browsing API v4 now consists of two APIs: the Lookup API and the Update API. The new Lookup API is the same as that of the previous Lookup API except that some constraints on request frequencies have been introduced. On the other hand, the Safe Browsing API v3 is now slightly modified and renamed as the Update API. Below, we discuss all the important changes in detail.

Blacklists: Safe Browsing blacklists are now identified using three attributes: `threatType`, `platformType`, `threatEntryType`. An example of these fields is: `Malware;Windows;URL` respectively. Every time a client requests an update of the local database, it has to precise these attributes.

Prefix length: The prefix length is now variable, varying from 4 bytes to 32 bytes (corresponding to a full hash). This is to reduce the risk of a popular web page colliding with a malicious URL.

Size constraints: Whenever the client requests for a local database update, it also specifies `maxUpdateEntries` that counts the total number of updates that the client can manage. Clients are only required to set size constraints if they have memory or bandwidth limitations.

Compression: Updates can now be sent in a compressed format. We note that compression was not supported in v3. If the client supports compression, it must indicate so using the field `supportedCompressions`. GSB currently supports Rice compression, which is similar to Golomb Coded Sets (*Cf.* Chapter 2 §2.7). We note that prefixes longer than 4 bytes are never compressed, and are served in the raw format instead.

Minimum wait time: When the server sends an update to the client, it may also indicate through a field `minimumWaitDuration` about the duration the client must wait before sending another update request. This is to prevent clients overloading the server with frequent update requests.

Full hash request: Clients are now encouraged to batch multiple prefixes into a single request to reduce the bandwidth usage. The server in return sends the hashes along with some caching duration to respect and may also return a minimum wait duration for the next request. The caching duration indicates the amount of time the hashes must be considered safe or unsafe. The goal is to prevent a client querying for a given prefix multiple times during a small duration. The minimum wait duration is required to avoid

[9] https://developers.google.com/safe-browsing/v4/

our DoS attacks towards the server, when full hash requests are made for different prefixes. Google states that:

> "*Caching is particularly important as it prevents traffic overload that could be caused by a hash prefix collision with a safe URL that receives a lot of traffic.*"

This measure is clearly meant to prevent our second pre-image attacks.

Privacy: No privacy specific changes have been introduced in version 4. For privacy, Google presents the following statement in this new version of Safe Browsing API:

> "*At no point does Google learn about the URLs you are examining. Google does learn the hash prefixes of URLs, but the hash prefixes don't provide much information about the actual URLs.*"

8.8 Summary

We have described several vulnerabilities in Google Safe Browsing and its siblings which allow an adversary to mount several DoS attacks. The attacks allow an adversary to either increase the false positive probability or force the service to believe that a target benign URL is possibly malicious, hence leading to a DoS. The situation is further exacerbated through our boomerang attacks. The Safe Browsing architecture (v3) further permits the adversary to simultaneously affect all the users of the service. In Google Safe Browsing, the back-end service attempts to implement a cache at the client side to reduce their servers' load. In fact, the digests stored are too short to prevent brute-force attacks. Increasing the digest length is a secure countermeasure, but the current data structure does not scale well with this modification and it can amplify the security issues by increasing the number of attacks on vulnerable websites, while also aggravating privacy issues.

With the release of Safe Browsing API v4, Google has taken some initiatives to prevent DoS attacks. However, the API v4 does not guarantee a better privacy than the one offered by API v3.

Chapter 9

Conclusions and Perspectives

This dissertation presents a security and privacy analysis of hash-based software applications. Our analysis takes a longitudinal approach as we consider a wide variety of applications of hashing in the context of security and privacy, ranging from the simplest use of hashing in pseudonymization to Bloom filters used in some sensitive software solutions and ultimately to an internet-scale application such as the Safe Browsing. All along our analysis, we explore the underlying issues, provide theoretical frameworks wherever possible and suggest both legacy-compatible countermeasures, as well as clean-slate solutions.

The work presented in this dissertation can be regrouped into three segments. In the first segment, we discuss the pitfalls of hashing for privacy and then present a new usecase of Bloom filters to achieve better privacy (Chapter 3 and Chapter 4). In the second segment, we study Bloom filters from a security perspective (Chapter 5). While, in the third segment, we study the security and privacy of Safe Browsing services (Chapter 7 and Chapter 8). In the rest of this chapter, we briefly recall our contributions to each segment and then present conclusions, perspectives and future work.

1. In the first segment of this dissertation, we discussed how plain hashing fails to ensure (pseudo)anonymization. And in the sequel, we discussed how Bloom filters can be used to build a privacy friendly solution to a real-world problem by coupling them with state-of-the-art cryptographic primitives.

Privacy applications of hashing and Bloom filters. Our analysis of plain hashing for pseudonymization clearly demonstrates that hashing gives a false sense of privacy and is bound to fail in most situations. More advanced applications of hashing to achieve stronger privacy notions such as anonymity set may also fail as such models do not generally consider the fact that an adversary may eventually acquire auxiliary information to break the anonymity. To conclude, any application of plain hashing in the context of privacy should correctly model the adversary's knowledge on the data being anonymized. In situations where the adversary's knowledge cannot be precisely modeled, data custodians must employ privacy-preserving techniques that work against most kinds of adversaries. A typical approach to this end should be to aim differential privacy. Bloom filter based

randomization techniques can prove to be useful in this regard. Indeed, some recent encouraging results propose a couple of differentially private solutions which have been shown to yield good utility in real-world situations.

We also employ Bloom filters to solve the problem of private membership queries to a public database. This is achieved by coupling them with private information retrieval protocols (PIRs). However, as seen in Chapter 4, even by using the most efficient PIR protocol due to Melchor *et al.* [MBFK16] on the SNAPCHAT database (49 MB), the protocol requires 4 mins. In comparison, the trivial solution which consists in sending the database requires roughly 20 seconds. This puts in evidence the fact that it is still hard to outperform the trivial solution.

Moreover, the Bloom filter based solution is an *ad hoc* approach to the problem, and hence there remains the open problem of designing dedicated and more efficient solutions to the problem of private membership query. Ideally, such a solution should not be built atop Bloom filters. This is because, if Bloom filters are used, the filter parameters should be so chosen such that the false positive probability should be very small, of the order of 2^{-80}. In fact, a (considerably) higher false positive probability will allow an adversary to find a false positive in reasonable time. For such a low false positive rate, the filter size should be large too. Hence, if the size of the items to be inserted in the Bloom filter is itself small, then the filter would eventually require more space than a simple list of all the items. To give an example, if entries in the database correspond to usernames, say 12 characters long, and if the false positive probability is 2^{-80}, then Bloom filters are in fact less economical than storing usernames per se.

2. In the second segment of this dissertation, we discussed the security of Bloom filters in adversarial settings. Our analysis reveals that the security implications of using Bloom filters is largely misunderstood. This is shown through a wide variety of software applications that are vulnerable to our attacks.

Security of Bloom filters. Our analysis of Bloom filters reveals that despite the far reaching attacks of Crosby *et al.* [CW03] and the several ensuing booster shots [KW11, AB12], non-cryptographic hash functions are still present in sensitive software solutions. If non-cryptographic hash functions are prevalent, cryptographic ones are often not used properly. After our work, we wish that software developers would become more cautious before truncating hashes or employing weak hash functions.

The study of algorithmic complexity attacks on hash tables revealed that an adversary may easily exploit the worst-case behavior of an algorithm or a data structure. However, despite previous results, we observe that the worst-case behavior of a Bloom filter has been largely ignored until now. Our results highlight that an adversary may easily force the filter to reach its worst case and may drastically impact the security of the concerned application. We also discover that Bloom hash tables perform much better in the worst-case than Bloom filters and hence have the potential to replace them. Additionally, Bloom hash tables have the advantage of being more memory efficient than Bloom filters.

There are three natural extensions to our work. First, studying other variants of Bloom filters, such as Stable Bloom filters [DR06], Spectral Bloom filters [CM03] and Bloomier filters [CKRT04]. Studying other variants of Bloom filters can be interesting to extend our adversary models. Many of these variants either add new functionalities or improve on the efficiency. For instance, Stable Bloom filters are in fact a variant of Counting Bloom filters for streaming data. Their purpose is to detect duplicates. However, since there is no way to store the entire history of a stream (which can be infinite), Stable Bloom filters continuously evict stale information to make room for more recent elements. Since stale information is evicted, a Stable Bloom filter introduces false negatives, which do not appear in traditional Bloom filters. Our pollution attacks should trivially extend to Stable Bloom filters. It would be interesting to see if it is also possible to mount deletion attacks. Apparently, this appears to be difficult to achieve since eviction as of such is random and hence cannot be controlled by the adversary. Another example is that of Bloomier filters [CKRT04] which can compute arbitrary functions. This data structure is much more involved than Bloom filters and hence its security analysis can prove to be a challenging task.

Second, investigating whether other algorithms and data structures are prone to similar algorithmic complexity attacks. This includes probabilistic counting algorithms [Fla04], similarity hashing algorithms such as MinHash [BCFM98], and hash table variants such as Cuckoo hashing [PR04] (essentially a strategy to resolve hash collisions). Probabilistic counting algorithms [Fla04] are very popular in computing statistics on large datasets with a reduced memory. Hashing (and the truncation that comes along) is the core mechanism. It will be interesting to analyze their existing implementations in an adversarial setting. This extends to other hash-based algorithms listed earlier. MinHash for instance computes the similarity between two files and gives the best results when the underlying hash functions are permutations. A preliminary investigation reveals that for the sake of efficiency, some real-world implementations do not employ permutations. This directs us to further study such bad implementations and their ensuing adversarial impact.

Third, studying if *extensible-output hash functions* [Nat14] can prevent the attacks described in this dissertation. Indeed, the ideal hash function for Bloom filters should be an efficient and secure keyed hash function with extensible output. Recently, the NIST has released the SHA-3 standard [Nat14]. It includes two extensible-output functions SHAKE-128 and SHAKE-256. These functions however have an unwanted feature. That is, when two different output lengths are chosen for a common message, the two outputs are closely related: the shorter output is a truncated version of the longer one. However, this issue should not be relevant when different messages are hashed using a fixed size. Nevertheless, these functions are not yet standardized[1] and hence we look forward to knowing if they are indeed secure and if they can be keyed. Additionally, we are also interested in knowing how good they perform with Bloom filters.

There is another detached line of work that we wish to pursue in the future. It stems from our study on how to learn filter parameters when they are not public. Our approach in

[1]Public comments on FIPS 202 http://csrc.nist.gov/groups/ST/hash/sha-3/documents/fips202_comments/Fluhrer_Comments_Draft_FIPS_202.pdf

this regard was to mount timing attacks. However, it is apparent that we have considered a simplified setting, where we ignored the noise that may originate from network round trips. We also ignored other unknown factors such as the remote system parameters that the attacker must replicate on his side. We hence wish to extend our findings to more general and realistic settings. This line of work is challenging. However, previous works on remote timing attacks on OpenSSL servers can prove to be a starting point [BB03, BT11]. If successful, this study will lead to important consequences in the field of password based authentication. Indeed, a user can thereby remotely learn the hash function used on the authentication server and hence can learn how well their passwords are protected. Until today, this information can only be obtained when the service provider publicly identifies the hash function (which is not so common for that matter) or when hackers publish leaked datasets of passwords.

3. The final segment of this manuscript discussed the security and privacy issues in Safe Browsing services. For privacy, we showed that Google and therefore Yandex are the only vendors that make sincere efforts to render their service privacy friendly. However, they fail to meet the guarantees that they advert. As for security, we concluded that the architecture in its current form is flawed as a motivated attacker can mount DoS attacks against both Safe Browsing servers and clients.

Security and privacy of Safe Browsing. It is quite apparent that all the major players in the Safe Browsing ecosystem (Google, Yandex and Mozilla Firefox) are aware of the security vulnerabilities in GSB and YSB. Nevertheless, some of them either do not care or they consider themselves to be well equipped to defend themselves against DoS attacks. Unfortunately, they completely disregard the issues on the client's side. By introducing the new GSB version 4 in June 2016, Google has however made some efforts to reduce the effect of our attacks. Some of these changes include the possibility to send compressed prefixes to the client using Golomb encoding. This reduces the required bandwidth. Time to wait for the next full hash query can also be forced onto the client to prevent request flooding in an attack scenario. Unlike in GSB version 3, where prefixes on the client's side are 32-bits in length, in GSB version 4, they can now be of variable length between 32 bits to 256 bits. This is to alleviate the effects of our second pre-image attacks on popular websites.

As for privacy, the situation is gravely grim because eventually none of the SB services respects the privacy of their users. Furthermore, GSB version 4 offers nothing new to ensure privacy. In face with the presented privacy threats, users have two solutions to safeguard themselves: disabling the SB services or improving them. Disabling SB exposes users' private information to malware and phishing threats. This is however the radical choice of Tor Browser developers.

The current architecture based on a local cache is clearly more privacy friendly than sending each URL to the server. This is essentially because information on the visited URL is leaked only when the URL is probably malicious. The probability that information on

a URL is leaked is equal to the total number of prefixes in the local cache divided by 2^{32}. Hence, as the number of prefixes in the local cache increases, the probability that an information leakage occurs also increases. On the one hand, including the cache may appear to reduce the privacy risks, while on the other hand, it also empowers Google to include any prefix of its choice in the cache and learn information about visits on specific web pages.

Hence, to improve these services, we need to guarantee privacy and accountability. The trivial PIR, *i.e.*, distributing the lists in clear to all the clients, is the only solution providing absolute privacy. Unfortunately, the trivial PIR comes with a memory/bandwidth overhead. This again raises the question of designing efficient and secure PIR protocols. Additionally, the trivial PIR also exposes weak and vulnerable websites to attackers. Considering these needs, it is evident that even in the case of SB services, we again need a very efficient and scalable private membership protocol.

From our analysis of GSB, we further propose a new relevant setup for the PIR problem: A user wants to retrieve an item from a database controlled by a third party without revealing his query. The setting here is new in the sense that, the user now additionally has some side information on the database (list of prefixes for instance). The open question that arises here is whether it possible to find usable PIR schemes by exploiting the side information.

Accountability is another reason why achieving the goal of the expected privacy guarantee is not a trivial task. Indeed, the vendors must also be made accountable for the URLs that they include in the blacklists. For which, it is required that the lists be signed and approved by a trusted third-party. To conclude, we need a secure, distributed and accountable blacklist management system to which a client can make private queries. To this end, some works addressing some of these issues already exist [MPC15,FCB15], where authors design a privacy-preserving collaborative blacklisting mechanism with the help of a semi-trusted third party. While the privacy of the collaborators is the main concern, their accountability is not assessed. However, these works can prove to be a promising starting point.

Another possible solution to the accountability problem could be a completely decentralized (no need of any trusted party) detection and management system of malicious URLs. Even though, Google currently consults different databases such as Phistank and `stopbadware.org` to feed its blacklist, the system is clearly not accountable and transparent to end users. A decentralized system based on the *Blockchain*[2] data structure could be a potential solution, where nodes in a collaborative network may reach consensus on a set of malicious URLs/digests and publish them in the blockchain. The blockchain can then be consulted by end users to check the status of a URL. While privacy of each node in the network may be required, a more important constraint however would be to ensure that the set of URLs/digests on which the consensus has been reached should not be publicly available in clear to malicious nodes. This is because any attacker may learn URLs or digests (from which the URLs can be reverse engineered) which are potentially vulnerable

[2] https://en.wikipedia.org/wiki/Blockchain_database

to future attacks. To this end, the project *Enigma* by Zyskind *et al.* [ZNP15] can prove to be a starting point. Enigma employs distributed hash tables for privacy-preserving storage and secure multi-party computations for privacy-preserving computations. Though, the Enigma framework is very generic, it should be possible to use it in a minimalistic way to meet the needs of GSB. The best way to evaluate such a system would be to incorporate the modifications into GSB and study the following aspects:

1. Privacy: Agreed upon malicious URLs or digests should not be learned by malicious adversaries. Additionally, anonymity of the participants may also be required.

2. Correctness: Low rate of false positives or false negatives.

3. Efficiency: How fast the consensus is reached? This can indirectly be measured by the throughput, *i.e.*, the number of malicious URLs detected every unit time. Memory considerations should also be studied. This is particularly important in the context of mobile devices.

As another future work, we wish to design a plugin for Firefox and Chrome to make users aware of the privacy issues associated with the use of SB services. This plugin will be an extension of the prototype that we developed to study the proposed mitigation measure in Chapter 7 (which consists in looking for Type I URLs for a given web page). Since our prototype is efficient, we expect the plugin to be so too. The plugin will be developed in JavaScript and will essentially rely on a fast crawler to recover URLs on a given domain. Depending on the crawled results and the ensuing analysis, users can be informed in real-time about the possibility that the SB server can learn a web page visited by a user. We note that the plugin will perform all the analysis locally on the client's side and hence will ensure that it does not entail any privacy leakage.

Part III

Back Matter

List of Figures

2.1 A hash table to represent a phone book. It contains three key-value pairs: ('Tom', '7953'), ('Dick', '7941') and ('Harry', '7832'). The colored cell represents a collision of the hash function on the keys 'Dick' and 'Harry'. A linked list is used to handle collisions. Keys are also stored along with the values to remove ambiguity for queries such as: What is the phone number of Dick? 12

2.2 A binary Merkle tree with 4 data blocks as leaves. Hash-00 = h(Data-1); Hash-01 = h(Data-2); Hash-0 = h(Hash-00||Hash-01) = $h(h$(Data-1)||h(Data-2)), where || is the usual concatenation. Other nodes in the tree are defined in a similar manner. 14

2.3 A Bloom filter with $m = 12$ and $k = 2$. 16

2.4 Updating the `count` matrix for a data stream using $d = 5$ hash functions and $n = 7$. The colored entries correspond to the collision of hash functions h_1 and h_2. 22

3.1 A one-to-one pseudonymization. 36

3.2 A many-to-one pseudonymization. id_1 and id_2 yield the same pseudonym d_i. 36

3.3 Gravatar URL generation from an email address. 39

3.4 Structure of a MAC address. 42

3.5 Number of OUIs per vendor (top 6). 44

3.6 Percentage of vendors of each type. 45

4.1 BLIP steps for a profile P with three items. The filter is built using 2 hash functions and a flip probability of p. Flipping of a bit b in the filter is shown in Step 2. 51

4.2 Steps in RAPPOR for the string `"I use RAPPOR"`. The filter is built using 2 hash functions and with a flip probability p. Flipping of a bit b in the filter is shown in Step 2. 52

4.3 A sample alerting website: https://lastpass.com. 55

4.4 Verification on a constraint channel. 61

5.1 Items x_1 and x_2 are inserted into the filter using 2 hash functions. While both y_1 and y_2 are false positives. Only y_2 defines a pre-image of x_2. This is because the first digest of y_1 corresponds to a bit occupied by x_1, while the second digest corresponds to a bit occupied by x_2. 73

5.2 Chosen-insertion adversary ($k = 2$). 73

5.3 False positive probability as a function of inserted items ($m = 3200$, $k = 4$ and $f_{\text{opt}} = 0.077$). 75
5.4 Crafted false positives: $w_H(\vec{z}) = 5$. The first digest $h_1(\cdot)$ corresponds to the left arrow for each item, while the second digest $h_2(\cdot)$ corresponds to the right arrow. 76
5.5 Cost of creating polluting URLs. 81
5.6 Cost of creating ghost URLs. The curve for $f = 2^{-5}$ is truncated as it took several hours to find a false positive when the filter occupation was less that 40%. 81
5.7 A root domain `root.com` with the page tree `main, main/tags, main/tags/app` is chosen. Once Scrapy recursively visits the decoys, the ghost page `main/tags/app/ghost` is considered as already visited. The ghost page thus remains hidden (hence a dotted arrow and a cross). 82
5.8 Polluting Dablooms. F represents the combined false positive probability (see Chapter 2). The last curve in black from left represents the false positive probability achieved when only the last filter is polluted. 83
5.9 Comparing the cost of evaluating a hash function and the cost of querying a Bloom filter, when built using the same hash function. Filter parameters are $k = 10, n = 10^6$, and item size is 32 bytes. 90
5.10 Bloom hash table data structure. 92
5.11 Memory comparison for Bloom filters (m) and Bloom hash tables (m'). 92
5.12 Domain of application of hash functions. 96

6.1 Status of ianfette.org on WOT. 106
6.2 WOT annotates results of a search query for the keyword 'malware bytes'. 107
6.3 Norton Safe Search for Chrome. 107
6.4 Norton Safe Search annotates results of a search query for the keyword 'malware download'. 108
6.5 Warning page displayed to the user for a web page hosting malware. 109
6.6 High level overview of Google Safe Browsing. 110
6.7 Google Safe Browsing API: Client's behavior flow chart. 113
6.8 Evolution of the index vector and the delta vector in a delta-coded prefix table. At step (1), the vectors are empty. The final state of the vectors is shown in step (5). 116
6.9 Diagnostic result for ianfette.org. 120
6.10 Diagnostic result for the application server that hosts ianfette.org. 120
6.11 The Safe Browsing ecosystem. 123

7.1 Safe Browsing à la PIR. 126
7.2 A simplified structure of Google Safe Browsing. The client makes a conditional query to the server when the URL creates a hit in the local copy. 126

7.3 A sample domain hierarchy for b.c. Colored nodes represent malicious resources. A real world example can be google.com for b.c; mail.google.com for a.b.c, analytics.google.com for d.b.c, and maps.google.com for e.b.c. 130
7.4 A sample domain hierarchy for b.c. Colored nodes are leaf URLs: a.b.c/1, a.b.c/2, a.b.c/3/3.1, a.b.c/3/3.2 and d.b.c. A real world example can be google.com for b.c; mail.google.com for a.b.c and analytics.google.com for d.b.c. 134
7.5 Distribution of URLs and decompositions on hosts from the two datasets. Figure (a) presents the distribution of URLs over hosts, while (b) presents its cumulative distribution. (c) shows the distribution of decompositions over hosts. (d), (e) and (f) respectively present the mean, minimum and maximum number of URL decompositions on the hosts. 135
7.6 The domain hierarchy of petsymposium.org. Leaf URLs are in blue. The dashed triangle represents the sub-hierarchy for petsymposium.org/2015. The URLs in the dashed rectangle create Type I collisions with petsymposium.org/2016. 138

8.1 The attack routine: An adversary forges malicious URLs and diffuses them. These URLs then get detected and indexed by the Google crawlers and eventually reach the clients through the database update. 149
8.2 Number of second pre-images found per URL in the Alexa list of 1M URLs. 153
8.3 Three possible topologies of the domains generated by an adversary are depicted. The lowest node in the graphs represent the root domain, while the other nodes correspond to the files or the directories created on the root domains. The topology (a) represents the case where a new domain is created for each prefix, (b) represents the domains generated by the same depth strategy, and (c) represents those corresponding to the increasing depth strategy. 155
8.4 An excerpt of the analysis by VirusTotal (http://virustotal.com/) of our phishing page. 159
8.5 The Safe Browsing diagnostic tool shows google.com as potentially malicious. Accessed on April 21, 2016. 160

List of Tables

2.1 A summary of the data structures presented in this chapter. a in the time complexity for hash tables and GCS denotes the average case. 25

3.1 Computation time for 2^{48} SHA-1 digests using our own hardware (the first two rows) and the benchmark results from `oclHashcat` (the remaining rows). The number of hashes per second provides the average perfomance of the cracking tool on the concerned hardware. Cells marked * are estimated results. We used the number of hashes per second to estimate the time required to hash 2^{48} MAC addresses. 43

3.2 Time to discriminate the top 6 vendors in the OUI list using ATI R9 280X GPU. 44

4.1 Analysis of 17 alerting websites (* result as on 22/12/2014). 56

4.2 Detailed analysis of `lastpass.com`. 57

4.3 Results for the leaked databases using SHA-1. Databases contain single data for a user, for instance Snapchat contains only username and ignores other auxiliary leaked information. The experimental results are shown with decreasing false positive probability f. 64

4.4 Cost for PSI protocol [DCT10] with 80 bits of security using SHA-1. 65

4.5 Cost for PSI-CA protocol [DCGT12] with 80 bits of security using SHA-1. 65

4.6 Summary of the results on Snapchat with $f = 2^{-32}$. 66

5.1 Comparative summary of our attacks for a filter of Hamming weight W. ℓ is the size of the digest generated by a cryptographic hash function. 78

5.2 Results of our experiments for filters of capacity 1 million and with varying number of hash functions, k. We present the expected and the observed number of false positives for 10 million random items. k_{est} shows the value estimated by the adversary using the relation: $k_{\text{est}} = -\log_2(\text{Observed \#false positives}/10^7)$. 88

5.3 Time to query a filter. 96

6.1 Lists provided by Google Safe Browsing. Information could not be obtained for cells marked with *. 110

6.2 Client's local database size for different prefix sizes. 117

6.3 Yandex blacklists. Information could not be obtained for cells marked with *. 122

7.1 Events leaking information in GSB and YSB. 128

7.2 Decompositions of https://conf.org/2016/cfp.html. 128
7.3 Max. and avg. values for URLs and domains with prefix size ℓ. 0^* represents a value close to 0. 130
7.4 An example with different possible collisions. The decomposition algorithm is the one described in Chapter 6. URLs g.a.b.c and a.b.c are "related" since they share two common decompositions, namely a.b.c/ and b.c./. 132
7.5 A sample URL on b.c with its 4 decompositions. 133
7.6 Our datasets. 135
7.7 Decompositions of PETS CFP URL. 138
7.8 Dataset used for inverting 32-bit prefixes. 139
7.9 Matches found with our datasets. 140
7.10 Distribution of prefixes as the number of full hashes per prefix. Collision with the Alexa list is also given. 141
7.11 A subset of URLs from the Alexa list creating multiple hits in the Google and Yandex database. 142

8.1 Distribution of the reply size in the number of full hashes per prefix for malware and phishing blacklists. 151
8.2 Prefixes and the domains in the Alexa list for which most second pre-images were found. A sample second pre-image is also provided. The search gave 160 second pre-images for 2 websites. 153
8.3 Same depth strategy: Sample second pre-images for popular web pages. 154
8.4 Increasing depth strategy: Sample second pre-images for popular web pages. 154
8.5 Comparative summary of the different strategies for generating malicious domains. The comparison is relative to a scenario without any attack on the service. The symbol '$\emptyset$' denotes that no change could be obtained by the strategy. We use the symbol '+' to indicate the relative magnitude of the different values. Adversary's complexity inversely relates to the level of parallelization that she can achieve. 156

Bibliography

[AB12] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A Fast Short-Input PRF. In *Progress in Cryptology - INDOCRYPT 2012*, Lecture Notes in Computer Science 7668, pages 489–508, Kolkata, India, December 2012. Springer.

[ABPH07] Paulo Sérgio Almeida, Carlos Baquero, Nuno M. Preguiça, and David Hutchison. Scalable Bloom Filters. *Inf. Process. Lett.*, 101(6):255–261, 2007.

[ACRF14] Jagdish Prasad Achara, Mathieu Cunche, Vincent Roca, and Aurélien Francillon. Short paper: Wifileaks: underestimated privacy implications of the access_wifi_state android permission. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 231–236, New York, NY, USA, 2014. ACM, ACM.

[AF13] Devdatta Akhawe and Adrienne Porter Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 257–272, 2013.

[AFRC14] Hazim Almuhimedi, Adrienne Porter Felt, Robert W. Reeder, and Sunny Consolvo. Your reputation precedes you: History, reputation, and the chrome malware warning. In *Tenth Symposium on Usable Privacy and Security, SOUPS 2014, Menlo Park, CA, USA, July 9-11, 2014*, pages 113–128, 2014.

[AGK12] Mohammad Alaggan, Sébastien Gambs, and Anne-Marie Kermarrec. BLIP: Non-interactive Differentially-Private Similarity Computation on Bloom filters. In *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium, SSS 2012, Toronto, Canada, October 1-4, 2012. Proceedings*, pages 202–216, 2012.

[AGMT15] Mohammad Alaggan, Sébastien Gambs, Stan Matwin, and Mohammed Tuhin. Sanitization of Call Detail Records via Differentially-Private Bloom Filter. In *Data and Applications Security and Privacy XXIX - 29th Annual IFIP WG 11.3 Working Conference, DBSec 2015, Fairfax, VA, USA, July 13-15, 2015, Proceedings*, pages 223–230, 2015.

[AIE16] AIEngine, 2016. https://bitbucket.org/camp0/aiengine.

[Ale15] Alexa 1M Global Sites. Online, 2015. http://bit.ly/1yhXcgL.

[AM04] Arvind Arasu and Gurmeet Singh Manku. Approximate Counts and Quantiles over Sliding Windows. In *Proceedings of the Twenty-third ACM*

SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France, pages 286–296, 2004.

[aol06] A Face Is Exposed for AOL Searcher No. 4417749, 2006. http://www.nytimes.com/2006/08/09/technology/09aol.html?_r=0.

[APD+10] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster. Building a Dynamic Reputation System for DNS. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 273–290, 2010.

[App10] Austin Appleby. smhasher - Test your hash functions., 2010. https://code.google.com/p/smhasher/.

[Att16] Attributor, Digimarc, 2016. http://www.digimarc.com.

[Bac02] Adam Back. Hashcash - A Denial of Service Counter-Measure. Technical report, Cypherspace, 2002.

[BB03] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, 2003.

[BBB12] Frank Breitinger, Harald Baier, and Jesse Beckingham. Security and implementation analysis of the similarity digest sdhash. In *First International Baltic Conference on Network Security & Forensics (NeSeFo)*, 2012.

[BC04] Steve Bellovin and William R. Cheswick. Privacy-Enhanced Searches Using Encrypted Bloom Filters. In *DIMACS/PORTIA Workshop on Privacy-Preserving Data Mining*, pages 274–285, Piscataway, NJ, USA, March 2004. DIMACS/PORTIA.

[BCCL07] Tian Bu, Jin Cao, Aiyou Chen, and Patrick P. C. Lee. A fast and compact method for unveiling significant patterns in high speed networks. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 6-12 May 2007, Anchorage, Alaska, USA*, pages 1893–1901, 2007.

[BCFM98] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise Independent Permutations. *Journal of Computer and System Sciences*, 60:327–336, 1998.

[BCMR02] John W. Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. Informed content delivery across adaptive overlay networks. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication - ACM SIGCOMM 2002*, pages 47–60, Pittsburgh, PA, USA, August 2002. ACM.

[Ber01] Daniel J. Bernstein. djbdns, 2001. http://cr.yp.to/djbdns.html.

[BGK+08] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel H. M. Smid, and Yihui Tang. On the false-positive rate of Bloom filters. *Inf. Process. Lett.*, 108(4):210–213, 2008.

[Big15] BigBlackList. Online, 2015. http://urlblacklist.com/.

[Bit12] Bitly Inc. dablooms, 2012. http://word.bitly.com/post/28558800777/dablooms-an-open-source-scalable-counting.

[BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. Technical Report 3986, RFC Editor, January 2005. Updated by RFCs 6874, 7320, http://www.ietf.org/rfc/rfc3986.txt.

[BLMM94] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). RFC 1738, RFC Editor, December 1994. https://www.ietf.org/rfc/rfc1738.txt.

[Blo70] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.

[BM05] Andrei Broder and Michael Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4), 2005.

[Bon13] Dominique Bongard. De-anonymizing Users of French Political Forums. Technical report, 0xcite LLC, Luxembourg, October 2013. http://archive.hack.lu/2013/dbongard_hacklu_2013.pdf.

[Boo05] C++ Library Extensions, 2005. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf.

[BPBBLP12] Udi Ben-Porat, Anat Bremler-Barr, Hanoch Levy, and Bernhard Plattner. On the Vulnerability of Hardware Hash Tables to Sophisticated Attacks. In *Networking (1)*, volume 7289 of *Lecture Notes in Computer Science*, pages 135–148. Springer, 2012.

[BPj14] BPjM. BPJM Modul. Online, 2014. http://bpjmleak.neocities.org/.

[BR09] Darrell Bethea and Michael K. Reiter. Data Structures with Unpredictable Timing. In *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 456–471. Springer, 2009.

[BT11] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, pages 355–371, 2011.

[BYW07] Noa Bar-Yosef and Avishai Wool. Remote Algorithmic Complexity Attacks against Randomized Hash Tables. In *International Conference on Security and Cryptography - SECRYPT 2007*, pages 117–124, Barcelona, Spain, July 2007. Springer Berlin Heidelberg.

[CDKL16] Mathieu Cunche, Levent Demir, Amrit Kumar, and Cédric Lauradoux. The Pitfalls of Hashing for Privacy, 2016. Submitted to IEEE Communications Surveys & Tutorials.

[CFG+78] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 59–65, New York, NY, USA, 1978. ACM.

[CGJ09] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In *IEEE Symposium on Security and Privacy (S&P) 2009*, pages 27–41, Oakland, California, USA, May 2009. IEEE Computer Society.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private Information Retrieval. In *Annual Symposium on Foundations of Computer Science, FOCS 1995*, 1995.

[CGN98] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords, 1998.

[Cha82] David Chaum. Blind Signatures for Untraceable Payments. In *Advances in Cryptology: Proceedings of CRYPTO '82, Santa Barbara, California, USA, August 23-25, 1982.*, pages 199–203, 1982.

[Cha04] Yan-Cheng Chang. Single Database Private Information Retrieval with Logarithmic Communication. In *Information Security and Privacy*, volume 3108. Springer Berlin Heidelberg, 2004.

[CJC10] Aiyou Chen, Yu Jin, and Jin Cao. Tracking Long Duration Flows in Network Traffic. In *INFOCOM 2010. 29th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 15-19 March 2010, San Diego, CA, USA*, pages 206–210, 2010.

[CKRT04] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 30–39, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[CKV10] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code. In *Proceedings of the 19th International Conference on World Wide Web*, WWW'10, pages 281–290, New York, NY, USA, 2010. ACM.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[CM03] Saar Cohen and Yossi Matias. Spectral Bloom Filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 241–252, New York, NY, USA, 2003. ACM.

[CM05] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[CMS99] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally Private Information Retrieval with Polylogarithmic Communication. In *Advances in Cryptology– EUROCRYPT '99*. Springer Berlin Heidelberg, 1999.

[Com15] Common Crawl. Online, 2015. http://commoncrawl.org/.

[Cro03] Scott A. Crosby. Denial of Service through Regular Expressions. In *USENIX Security Symposium: Work-In-Progress Reports*, page 1, Washington, USA, December 2003. USENIX Association.

[CW77] Larry Carter and Mark N. Wegman. Universal Classes of Hash Functions (Extended Abstract). In *ACM Symposium on Theory of Computing - STOC*, pages 106–112, Boulder, CO, USA, May 1977. ACM.

[CW03] Scott A. Crosby and Dan S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security Symposium*, Lecture Notes in Computer Science 7668, pages 3–3, Washington, USA, December 2003. USENIX Association.

[Dan12] Quynh Dang. Recommendation for Applications Using Approved Hash Algorithms. Technical Report SP 800-107 Revision 1, National Institute of Standards & Technology, August 2012.

[DBC+14] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and Xiaofeng Wang. The Tangled Web of Password Reuse. In *Network and Distributed System Security Symposium, NDSS 2014*, 2014.

[DCGT12] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and private computation of cardinality of set intersection and union. In *Cryptology and Network Security*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012.

[DCT10] Emiliano De Cristofaro and Gene Tsudik. Practical Private Set Intersection Protocols with Linear Complexity. In *Proceedings of the 14th International Conference on Financial Cryptography and Data Security*, 2010.

[DCT12] Emiliano De Cristofaro and Gene Tsudik. Experimenting with Fast Private Set Intersection. In *Trust and Trustworthy Computing*. Springer Berlin Heidelberg, 2012.

[DCW13] Changyu Dong, Liqun Chen, and Zikai Wen. When Private Set Intersection Meets Big Data: An Efficient and Scalable Protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS'13, 2013.

[Deu96] L. P. Deutsch. RFC 1951: DEFLATE compressed data format specification version 1.3, May 1996. ftp://ftp.internic.net/rfc/rfc1951.txt.

[dM14] Xavier de Carné de Carnavalet and Mohammad Mannan. From Very Weak to Very Strong: Analyzing Password-Strength Meters. In *Network and Distributed System Security Symposium, NDSS 2014*, 2014.

[DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, pages 265–284, 2006.

[DN93] Cynthia Dwork and Moni Naor. Pricing via Processing or Combatting Junk Mail. In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '92, pages 139–147, London, UK, UK, 1993. Springer-Verlag.

[Dns13] DNS Census. Online, 2013. https://dnscensus2013.neocities.org/.

[DR06] Fan Deng and Davood Rafiei. Approximately Detecting Duplicates for Streaming Data Using Stable Bloom Filters. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 25–36, New York, NY, USA, 2006. ACM.

[DSMRY09] Dana Dachman-Soled, Tal Malkin, Mariana Raykova, and Moti Yung. Efficient Robust Private Set Intersection. In *Applied Cryptography and Network Security*. Springer Berlin Heidelberg, 2009.

[Dwo06] Cynthia Dwork. Differential privacy. In *33rd International Colloquium on Automata, Languages and Programming, part II (ICALP 2006)*, volume 4052 of *Lecture Notes in Computer Science*, pages 1–12, Venice, Italy, July 2006. Springer Verlag.

[ECH08] Serge Egelman, Lorrie Faith Cranor, and Jason I. Hong. You've been warned: an empirical study of the effectiveness of web browser phishing warnings. In *Proceedings of the 2008 Conference on Human Factors in Computing Systems, CHI 2008, 2008, Florence, Italy, April 5-10, 2008*, pages 1065–1074, 2008.

[EDG14] Tariq Elahi, George Danezis, and Ian Goldberg. PrivEx: Private Collection of Traffic Statistics for Anonymous Communication Networks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1068–1079, 2014.

[EO08] Gunes Ercal-Ozkaya. *Routing in Random Ad-Hoc Networkds: Provably Better than Worst-case*. PhD thesis, University of California at Los Angeles, 2008.

[EPK14] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1054–1067, 2014.

[epr02] Directive concerning the processing of personal data and the protection of privacy in the electronic communications sector, 2002. http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:32002L0058:en:HTML.

[ES13] Serge Egelman and Stuart E. Schechter. The importance of being earnest [in security warnings]. In *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, pages 52–59, 2013.

[Euc15] Euclid Inc. Euclid Analytics - Privacy Statement. Online, Accessed on 15/05/15, 2015. http://euclidanalytics.com/privacy/statement/.

[Fac12] Facebook Inc. Link Shim. http://on.fb.me/1he2yEB, 2012.

[FCAB00] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000.

[FCB15] Julien Freudiger, Emiliano De Cristofaro, and Alejandro E. Brito. Controlled data sharing for collaborative predictive blacklisting. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, pages 327–349, 2015.

[Fel68] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, January 1968.

[FKD+13] S. Farrell, D. Kutscher, C. Dannewitz, B. Ohlman, A. Keranen, and P. Hallam-Baker. Naming Things with Hashes. RFC 6920, RFC Editor, April 2013. http://tools.ietf.org/html/rfc6920.

[FKP10] Márk Félegyházi, Christian Kreibich, and Vern Paxson. On the Potential of Proactive Domain Blacklisting. In *LEET*, 2010.

[Fla04] Philippe Flajolet. Theory and Practice of Probabilistic Counting Algorithms (Abstract of Invited Talk). In *Workshop on Analytic Algorithmics and Combinatorics - ANALC 2004*, page 152, New Orleans, LA, USA, January 2004. SIAM.

[FNP04] MichaelJ. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient Private Matching and Set Intersection. In *Advances in Cryptology - EUROCRYPT 2004*. Springer Berlin Heidelberg, 2004.

[GAK14] Neha Gupta, Anupama Aggarwal, and Ponnurangam Kumaraguru. bit.ly/can-do-better, February 2014. Poster presented at Security Privacy Symposium - SPS 2014.

[Gig16] Giga Alert, 2016. http://www.gigaalert.com/products.php.

[GK04] Michael Greenwald and Sanjeev Khanna. Power-Conserving Computation of Order-Statistics over Sensor Networks. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, pages 275–285, 2004.

[GKL14] Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. (Un)Safe Browsing. Research Report RR-8594, INRIA, September 2014.

[GKL15] Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. The Power of Evil Choices in Bloom Filters. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25*, 2015.

[GKL16] Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. A Privacy Analysis of Google and Yandex Safe Browsing. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 25-July 1*, 2016.

[GLLY10] Deke Guo, Yunhao Liu, XiangYang Li, and Panlong Yang. False Negative Problem of Counting Bloom Filter. *Knowledge and Data Engineering, IEEE Transactions on*, 22(5):651–664, May 2010.

[Goh03] Eu-Jin Goh. Secure Indexes. Cryptology ePrint Archive, Report 2003/216, 2003. http://eprint.iacr.org/2003/216/.

[Gol66] Solomon W. Golomb. Run-length encodings (Corresp.). *IEEE Transactions on Information Theory*, 12(3), 1966.

[Gol07] I. Goldberg. Improving the Robustness of Private Information Retrieval. In *IEEE Symposium on Security and Privacy, S&P '07*, 2007.

[Goo08] Google Inc. We knew the web was big... http://bit.ly/1P4jKwe, 2008.

[Goo12] Google Inc. Google Safe Browsing Service in Mozilla Firefox Version 3. http://bit.ly/1igzX4v, April 2012.

[Goo14a] Dan Goodin. Poorly anonymized logs reveal NYC cab drivers' detailed whereabouts, June 2014. Accessed 21-07-16.

[Goo14b] Google. Google Chrome Privacy Notice. https://www.google.com/intl/en/chrome/browser/privacy/, November 2014.

[Goo14c] Google Inc. Google Transparency Report. Technical report, Google, June 2014. https://bit.ly/1A72tdQ.

[Goo15a] Google Inc. Safe Browsing API. https://developers.google.com/safe-browsing/, 2015.

[Goo15b] Google Inc. Safe Browsing Lookup API. https://developers.google.com/safe-browsing/lookup_guide, 2015.

[GR05] Craig Gentry and Zulfikar Ramzan. Single-Database Private Information Retrieval with Constant Communication Rate. In *ICALP*, 2005.

[GSS+14] Arthur Gervais, Reza Shokri, Adish Singla, Srdjan Capkun, and Vincent Lenders. Quantifying Web-Search Privacy. In *ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 966–977. ACM, 2014.

[GXT+08] Sharon Goldberg, David Xiao, Eran Tromer, Boaz Barak, and Jennifer Rexford. Path-quality monitoring in the presence of adversaries. In *International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2008*, pages 193–204, Annapolis, MD, USA, June 2008. ACM.

[HA99] Bernardo A. Huberman and Lada A. Adamic. Internet: Growth dynamics of the World-Wide Web. *Nature*, 401(6749):131–131, 1999.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols? In *NDSS*, 2012.

[Her16] Heritrix, 2016. https://webarchive.jira.com/wiki/display/Heritrix/Heritrix.

[HJN+11] Ling Huang, Anthony D. Joseph, Blaine Nelson, Benjamin I. P. Rubinstein, and J. D. Tygar. Adversarial machine learning. In *ACM Workshop on Security and Artificial Intelligence, AISec 2011, , 21, 2011*, pages 43–58, Chicago, IL, USA, October 2011. ACM.

[HL08] Carmit Hazay and Yehuda Lindell. Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries. In *Theory of Cryptography*. Springer Berlin Heidelberg, 2008.

[HLCX11] Guanyao Huang, Ashwin Lall, Chen-Nee Chuah, and Jun (Jim) Xu. Uncovering Global Icebergs in Distributed Streams: Results and Implications. *J. Network Syst. Manage.*, 19(1):84–110, 2011.

[HN12] Carmit Hazay and Kobbi Nissim. Efficient Set Operations in the Presence of Malicious Adversaries. *Journal of Cryptology*, 25(3), 2012.

[HW06] Susan Hohenberger and StephenA. Weis. Honest-Verifier Private Disjointness Testing Without Random Oracles. In *Privacy Enhancing Technologies*, Lecture Notes in Computer Science, pages 277–294. Springer Berlin Heidelberg, 2006.

[IEE16] IEEE, 2016. http://standards-oui.ieee.org/oui.txt.

[Jen96] Robert Jenkins. A Hash Function for Hash Table Lookup, 1996. http://www.burtleburtle.net/bob/hash/doobs.html.

[JGW08] Paul F. Farrell Jr., Simson L. Garfinkel, and Douglas White. Practical applications of bloom filters to the NIST RDS and hard drive triage. In *Twenty-Fourth Annual Computer Security Applications Conference, ACSAC 2008, Anaheim, California, USA, 8-12 December 2008*, pages 13–22, 2008.

[JKPT07] Rosie Jones, Ravi Kumar, Bo Pang, and Andrew Tomkins. "I know what you did last summer": query logs and user privacy. In *ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007*, pages 909–914. ACM, 2007.

[JL09] Stanisław Jarecki and Xiaomin Liu. Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection. In *Theory of Cryptography*. Springer Berlin Heidelberg, 2009.

[JR13] Ari Juels and Ronald L. Rivest. Honeywords: making password-cracking detectable. In *ACM SIGSAC Conference on Computer and Communications Security, CCS'13*, 2013.

[KAPK13] Georgios Kontaxis, Elias Athanasopoulos, Georgios Portokalidis, and Angelos D. Keromytis. SAuth: protecting user accounts from password database leaks. In *ACM SIGSAC Conference on Computer and Communications Security, CCS'13*, 2013.

[KBC+05] George Kollios, John W. Byers, Jeffrey Considine, Marios Hadjieleftheriou, and Feifei Li. Robust Aggregation in Sensor Networks. *IEEE Data Eng. Bull.*, 28(1):26–32, 2005.

[Ken14] Paul D. Kennedy. Google's Safe Browsing Service is Killing Your Privacy. Online, 2014. http://bit.ly/1P2EEMk.

[Ker11] Florian Kerschbaum. Public-Key Encrypted Bloom Filters with Applications to Supply Chain Integrity. In *Data and Applications Security and Privacy XXV - 25th Annual IFIP WG 11.3 Conference, DBSec 2011*, Lecture Notes in Computer Science 6818, pages 60–75, Richmond, VA, USA, July 2011. Springer.

[KL14] Amrit Kumar and Cédric Lauradoux. Private Password Auditing - Short Paper. In *Technology and Practice of Passwords - International Conference on Passwords, PASSWORDS'14, Trondheim, Norway, December 8-10, 2014, Revised Selected Papers*, 2014.

[KL15] Amrit Kumar and Cédric Lauradoux. A Survey of Alerting Websites: Risks and Solutions. In *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28*, 2015.

[KLL14] Amrit Kumar, Pascal Lafourcade, and Cédric Lauradoux. Short Paper: Performances of cryptographic accumulators. In *39th IEEE Conference on Local Computer Networks, LCN 2014, Edmonton, AB, Canada, 8-11 September, 2014*, 2014.

[KLL16] Amrit Kumar, Cédric Lauradoux, and Pascal Lafourcade. Bloom Filters in Adversarial Settings, 2016. Submitted to ACM Transactions on Privacy and Security (TOPS).

[KM08] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. *Random Struct. Algorithms*, 33(2):187–218, 2008.

[KM16] Neal Koblitz and Alfred J. Menezes. Cryptocash, cryptocurrencies, and cryptocontracts. *Designs, Codes and Cryptography*, 78(1):87–102, 2016.

[KO97] E. Kushilevitz and R. Ostrovsky. Replication is Not Needed: Single Database, Computationally-private Information Retrieval. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, 1997.

[KS05] Lea Kissner and Dawn Song. Privacy-Preserving Set Operations. In *Advances in Cryptology – CRYPTO 2005*. Springer Berlin Heidelberg, 2005.

[KW11] Alexander Klink and Julian Wälde. Multiple Implementations Denial-of-Service via Hash Algorithm Collision. Technical Report oCERT advisory 2011–003, Open Source Computer Security Incident Response Team, march 2011.

[LHK+16] Frank Li, Grant Ho, Eric Kuan, Yuan Niu, Lucas Ballard, Kurt Thomas, Elie Bursztein, and Vern Paxson. Remedying Web Hijacking: Notification Effectiveness and Webmaster Comprehension. In *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, pages 1009–1019, 2016.

[Lip05] Helger Lipmaa. An Oblivious Transfer Protocol with Log-Squared Communication. In *ISC*, 2005.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[MBD+07] Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven D. Gribble, and Henry M. Levy. SpyProxy: Execution-based Detection of Malicious Web Content. In *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*, 2007.

[MBFK16] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR : Private Information Retrieval for Everyone. *PoPETs*, 2016(2):155–174, 2016.

[MBGL06] Alexander Moshchuk, Tanya Bragin, Steven D. Gribble, and Henry M. Levy. A Crawler-based Study of Spyware in the Web. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2006, San Diego, California, USA*, 2006.

[McA15a] McAfee. McAfee Site Advisor. http://www.siteadvisor.com/, 2015.

[McA15b] McAfee. McAfee Site Advisor FAQ. https://kc.mcafee.com/corporate/index?page=content&id=KB73457, 2015.

[McA15c] McAfee. McAfee Site Advisor Live. http://home.mcafee.com/store/siteadvisor-live, 2015.

[McI99] M. D. McIlroy. A Killer Adversary for Quicksort. *Softw. Pract. Exper.*, 29(4):341–344, apr 1999.

[MDC15] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient private statistics with succinct sketches. *CoRR*, abs/1508.06110, 2015.

[ME12] ABM Musa and Jakob Eriksson. Tracking unmodified smartphones using wi-fi monitors. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, pages 281–294, New York, NY, USA, 2012. ACM, ACM.

[Mer88] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, London, UK, UK, 1988. Springer-Verlag.

[MG08] C.A. Melchor and P. Gaborit. A fast private information retrieval protocol. In *Information Theory, 2008. ISIT 2008. IEEE International Symposium on*, 2008.

[Mic11a] Microsoft. SmartScreen Application Reputation. http://blogs.msdn.com/b/ie/archive/2011/05/17/smartscreen-174-application-reputation-in-ie9.aspx, 2011.

[Mic11b] Microsoft. SmartScreen: Authenticode Code Signing. http://blogs.msdn.com/b/ieinternals/archive/2011/03/22/authenticode-code-signing-for-developers-for-file-downloads-building-smartscreen-application-reputation.aspx, 2011.

[Mic16] Microsoft. Windows Logo. https://msdn.microsoft.com/en-us/windows/dd203105.aspx, 2016.

[MKD+02] J. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, A. van Hoff, and D. Hellerstein. Delta encoding in HTTP. RFC 3229, RFC Editor, January 2002. http://tools.ietf.org/html/rfc3229.

[MLRN15] Tommi Meskanen, Jian Liu, Sara Ramezanian, and Valtteri Niemi. Private Membership Test for Bloom Filters. In *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*, pages 515–522, 2015.

[Moc87] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (INTERNET STANDARD), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2673, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604.

[MPC15] Luca Melis, Apostolos Pyrgelis, and Emiliano De Cristofaro. Building and measuring privacy-preserving predictive blacklists. *CoRR*, abs/1512.04114, 2015.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[MR04] Jelena Mirkovic and Peter L. Reiher. A taxonomy of DDoS attack and DDoS defense mechanisms. *Computer Communication Review*, 34(2):39–53, 2004.

[MT07] Frank McSherry and Kunal Talwar. Mechanism Design via Differential Privacy. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), October 20-23, 2007, Providence, RI, USA, Proceedings*, pages 94–103, 2007.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., 1st edition, 1996.

[Nak11] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. *www.bitcoin.org*, 2011. http://fastbull.dl.sourceforge.net/project/bitcoin/Design%20Paper/bitcoin.pdf/bitcoin.pdf.

[Nat12] National Institute of Standards and Technology. Secure Hash Standard (SHS). Technical Report FIPS PUB 180-4, National Institute of Standards & Technology, march 2012.

[Nat14] National institute of standards and technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical Report FIPS PUB 202, National Institute of Standards & Technology, May 2014. Draft.

[Nat16] National Institute of Standards and Technology, 2016. http://www.nsrl.nist.gov/.

[Naz09] Jose Nazario. PhoneyC: A Virtual Client Honeypot. In *Proceedings of the 2nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, LEET'09, Berkeley, CA, USA, 2009. USENIX Association.

[NGSA08] Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary R. Anderson. Synopsis Diffusion for Robust Aggregation in Sensor Networks. *TOSN*, 4(2), 2008.

[NK09] Ryo Nojima and Youki Kadobayashi. Cryptographically Secure Bloom-Filters. *Transactions on Data Privacy*, 2(2):131–139, 2009.

[NMS+14] Nick Nikiforakis, Federico Maggi, Gianluca Stringhini, M. Zubair Rafique, Wouter Joosen, Christopher Kruegel, Frank Piessens, Giovanni Vigna, and Stefano Zanero. Stranger danger: exploring the ecosystem of ad-based URL shortening services. In *International World Wide Web Conference, WWW '14*, pages 51–62, Seoul, Republic of Korea, April 2014. ACM.

[NS05] Arvind Narayanan and Vitaly Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *ACM Conference on Computer and Communications Security, CCS 2005*, 2005.

[NY15] Moni Naor and Eylon Yogev. Bloom Filters in Adversarial Environments. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 565–584, 2015.

[Nyb96] Kaisa Nyberg. Fast Accumulated Hashing. In *Fast Software Encryption - FSE 1996*, volume 1039 of *Lecture Notes in Computer Science*, Cambridge, UK, February 1996. Springer.

[OG12] Femi Olumofin and Ian Goldberg. Revisiting the Computational Practicality of Private Information Retrieval. In *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 2012.

[ON10] Christopher Olston and Marc Najork. Web Crawling. *Foundations and Trends in Information Retrieval*, 4(3):175–246, 2010.

[opi10] Opinion 2/2010 on online behavioral advertising, 2010. http://ec.europa.eu/justice/policies/privacy/docs/wpdocs/2010/wp171_en.pdf.

[opi11] Opinion 16/2011 on EASA/IAB Best Practice Recommendation on Online Behavioural Advertising, 2011. http://ec.europa.eu/justice/data-protection/article-29/documentation/opinion-recommendation/files/2011/wp188_en.pdf.

[opi14] Opinion 05/2014 on Anonymisation Techniques, 2014. http://ec.europa.eu/justice/data-protection/index_en.htm.

[Pax98] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 3–3, Berkeley, CA, USA, 1998. USENIX Association.

[Pes98] Alexander Peslyak. Designing and Attacking Port Scan Detection Tools. *Phrack Magazine*, 8(453):13, July 1998. http://phrack.org/issues/53/13.html#article.

[PK00] Andreas Pfitzmann and Marit Köhntopp. Anonymity, Unobservability, and Pseudonymity - A Proposal for Terminology. In *International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *Lecture Notes in Computer Science*, pages 1–9, Berkeley, CA, USA, July 2000. Springer.

[PMRM08] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All Your iFRAMEs Point to Us. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

[PMW+09] Bryan Parno, Jonathan M. McCune, Dan Wendlandt, David G. Andersen, and Adrian Perrig. CLAMP: Practical Prevention of Large-Scale Data Leaks. In *IEEE Symposium on Security and Privacy - S&P 2009*, 2009.

[PPM12] Antonis Papadogiannakis, Michalis Polychronakis, and Evangelos P. Markatos. Tolerating Overload Attacks Against Packet Capturing Systems. In *USENIX Annual Technical Conference*, pages 197–202, Boston, MA, USA, June 2012. USENIX Association.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *J. Algorithms*, 51(2):122–144, 2004.

[Pro12] Niels Provos. Safe Browsing - Protecting Web Users for 5 Years and Counting. http://bit.ly/208ra6P, 2012.

[PRUV10] Andrea Pietracaprina, Matteo Riondato, Eli Upfal, and Fabio Vandin. Mining Top-K Frequent Itemsets Through Progressive Sampling. *Data Min. Knowl. Discov.*, 21(2):310–326, 2010.

[PSS10] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, Hash-, and Space-efficient Bloom Filters. *J. Exp. Algorithmics*, 14, 2010.

[PW07] Tom Preston-Werner. Gravatar, 2007. https://en.gravatar.com/site/implement/.

[Rou10] Vassil Roussev. Data fingerprinting with similarity digests. In *Advances in Digital Forensics VI - Sixth IFIP WG 11.9 International Conference on Digital Forensics, Hong Kong, China, January 4-6, 2010, Revised Selected Papers*, pages 207–226, 2010.

[RS98] Martin Raab and Angelika Steger. "Balls into Bins" - A Simple and Tight Analysis. In *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*, RANDOM '98, pages 159–170, London,UK, 1998. Springer-Verlag.

[RW98] Alex Rousskov and Duane Wessels. Cache digests. In *Computer Networks and ISDN Systems*, pages 22–23, 1998.

[RZ06] Matthew Roughan and Yin Zhang. Secure Distributed Data-Mining and its Application to Large-Scale Network Measurements. *Computer Communication Review*, 36(1):7–14, 2006.

[SB07] S. Joshua Swamidass and Pierre Baldi. Mathematical Correction for Fingerprint Similarity Measures to Improve Chemical Retrieval. *Journal of Chemical Information and Modeling*, 47(3):952–964, 2007.

[SC07] Radu Sion and Bogdan Carbunar. On the Practicality of Private Information Retrieval. In *Proceedings of the Network and Distributed System Security Symposium – NDSS 2007*, San Diego, CA, USA, February 2007. The Internet Society.

[Sch05] E. E. Schultz. Denial-of-Service Attack. In H. Bigdoli, editor, *Handbook of Information Security*, volume 3. Wiley, December 2005.

[Scr16] Scrapy, 2016. http://scrapy.org/.

[SCZ11] Xiaoshan Sun, Liang Cheng, and Yang Zhang. A Covert Timing Channel via Algorithmic Complexity Attacks: Design and Analysis. In *IEEE International Conference on Communications, ICC 2011*, pages 1–5. IEEE, June 2011.

[SD02] Andrei Serjantov and George Danezis. Towards an Information Theoretic Metric for Anonymity. In *Privacy Enhancing Technologies, Second International Workshop, PET 2002*, volume 2482 of *Lecture Notes in Computer Science*, pages 41–53, San Francisco, CA, USA, April 2002. Springer.

[SP06] Joshua Spiegel and Neoklis Polyzotis. Graph-Based Synopses for Relational Selectivity Estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 205–216, 2006.

[SPG13] Ashkan Soltani, Andrea Peterson, and Barton Gellman. NSA uses Google cookies to pinpoint targets for hacking. *The Washington Post*, 2013.

[SRZ+10] Mikko Särelä, Christian Esteve Rothenberg, András Zahemszky, Pekka Nikander, and Jörg Ott. BloomCasting: Security in Bloom Filter Based Multicast. In *Nordic Conference on Secure IT Systems, NordSec 2010*, volume 7127 of *Lecture Notes in Computer Science*, pages 1–16, Espoo, Finland, 2010. Springer.

[Ste16] Jens Steube. hashcat advanced password recovery, 2016. http://hashcat.net.

[SVG10] Osman Salem, Sandrine Vaton, and Annie Gravey. A Scalable, Efficient and Informative Approach for Anomaly-based Intrusion Detection Systems: Theory and Practice. *Int. J. Netw. Manag.*, 20(5):271–293, September 2010.

[Sym15a] Symantec. Norton Safe Search. http://nortonsafe.search.ask.com/, 2015.

[Sym15b] Symantec. Norton Safe Web. https://safeweb.norton.com/, 2015.

[The15] Inc. The Radicati Group. Email statistics report, 2015-2019, 2015. www.radicati.com/wp/.../Email-Statistics-Report-2015-2019-Executive-Summary.pdf.

[Til14] Richard B. Tilley. Blackhash software, 2014. http://16s.us/software/Blackhash.

[TLP+16] Sandeep Tamrakar, Jian Liu, Andrew Paverd, Jan-Erik Ekberg, Benny Pinkas, and N. Asokan. The circle game: Scalable private membership test using trusted hardware. *CoRR*, abs/1606.01655, 2016. http://arxiv.org/abs/1606.01655.

[TM12] R. Joshua Tobin and David Malone. Hash pile ups: Using collisions to identify unknown hash functions. In *7th International Conference on Risks and Security of Internet and Systems, CRiSIS 2012, Cork, Ireland, October 10-12, 2012*, pages 1–6, 2012.

[TRL12] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Communications Surveys and Tutorials*, 14(1):131–155, 2012.

[usj06] Privacy Technology Focus Group: Final Report and Recommendations, 2006. http://www.it.ojp.gov/documents/privacy_technology_focus_group_full_report.pdf.

[VC05] Jaideep Vaidya and Chris Clifton. Secure set intersection cardinality with application to association rule mining. *J. Comput. Secur.*, 13(4):593–622, July 2005.

[Ved15] Daniel Veditz. Personal Communication, 2015. Security team at Mozilla Firefox.

[Ver15] Verisign Inc. http://verisigninc.com/en_US/innovation/dnib/index.xhtml, 2015.

[VSGB05] Shobha Venkataraman, Dawn Xiaodong Song, Phillip B. Gibbons, and Avrim Blum. New streaming algorithms for fast detection of superspreaders. In *Network and Distributed System Security Symposium, NDSS 2005*, San Diego, CA, USA, February 2005. The Internet Society.

[WBJ+06] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Samuel T. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *NDSS*, 2006.

[Wes04] Duane Wessels. *Squid: The Definitive Guide* . O'Reilly Media, 2004.

[WOT15a] WOT Services Ltd. Privacy Policy. https://www.mywot.com/en/privacy, 2015.

[WOT15b] WOT Services Ltd. Web of Trust. https://www.mywot.com, 2015.

[Yan16] Yandex. Yandex Safe Browsing. http://api.yandex.com/safebrowsing/, 2016.

[YM12] J. Yao and W. Mao. SMTP Extension for Internationalized Email. RFC 6531 (Proposed Standard), February 2012. http://www.ietf.org/rfc/rfc6531.txt.

[Zel15] Zeltser Security Corp. https://zeltser.com/malicious-ip-blocklists/, 2015.

[ZNP15] Guy Zyskind, Oz Nathan, and Alex Pentland. Enigma: Decentralized Computation Platform with Guaranteed Privacy. *CoRR*, abs/1506.03471, 2015. http://arxiv.org/abs/1506.03471.

[ZSS+04] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick Duffield, and Carsten Lund. Online Identification of Hierarchical Heavy Hitters: Algorithms, Evaluation, and Applications. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, pages 101–114, New York, NY, USA, 2004. ACM.