NEXT GENERATION PASSWORD-BASED AUTHENTICATION SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Rahul Chatterjee

August 2019

NEXT GENERATION PASSWORD-BASED AUTHENTICATION SYSTEMS

Rahul Chatterjee, Ph.D.

Cornell University 2019

Passwords, despite being the primary means for users to authenticate on the Web or to a computing device, are marred with several usability and security problems: users nowadays have too many accounts and passwords to remember; typing passwords correctly can be cumbersome, particularly on touch screen devices. As a result, users often pick simple, easy-to-remember, and easy-to-type passwords and reuse them across different websites. Simple passwords, unfortunately, are also easy-to-guess. Reused passwords can put all of a user's accounts at risk if any of them are compromised.

In this dissertation, I show how to improve the state of passwords and password-based authentication (PBA) systems by incorporating knowledge of real-world password distributions. I identify three challenges faced by current passwords and PBA systems. Using a combination of empirical and analytical methods, I first distill out the details of those problems and then use that knowledge to guide building the next generation PBA systems that provide better usability and security.

First, to help users deal with too many account passwords, I design NoCrack, a secure password vault system (also called password manager) that uses honey encryption to encrypt user passwords under a master password. Honey encryption provides NoCrack's vault ciphertexts with a novel property: decryption with any incorrect master password will output decoy but plausible-looking sets of passwords. Therefore, if an attacker tries to decrypt a NoCrack's vault ciphertext with several guesses for the master password, the attacker does not immediately

learn the correct master password even if it is included in the list of guesses. To learn which of the decrypted passwords are real, the attacker has to try them online, which can be relatively slow, potentially detectable by the websites for which the user has an account, and also subject to website rate-limiting for too many incorrect password submissions.

Besides having too many passwords, users often make mistakes while typing passwords, and, in current settings, login is rejected if the entered password is not exactly what is used during registration. This is annoying and counter-productive for legitimate users. Via studies conducted on Amazon Mechanical Turk and with Dropbox's production login infrastructure, I measured the extent that password typos cause a usability burden. I showed how to design PBA systems that can tolerate typos without degrading the security of passwords.

Finally, due to billions of breached passwords and rampant password reuse habits, credential stuffing attacks have become a serious threat to password security: an attacker can compromise a user account by simply trying the password of that user stolen from other websites. To prevent such attacks, some third party web services have started providing APIs for checking if a user's password is present in a leaked set of passwords. I give a framework to analyze the security requirements of such compromised credentials checking (C3) services. I go on to provide new C3 protocols that provide a better security/bandwidth trade-off.

BIOGRAPHICAL SKETCH

Rahul received his Bachelors of Technology (B.Tech) degree from Indian Institute of Technology (IIT) Kharagpur in 2012. After graduation, he worked for a year in a startup on high-frequency trading in Bangalore. Working on the startup was very exciting, at the beginning: His code was affecting the real-world exchanges across the globe (though not always in the intended way). However, soon he realized he was missing the opportunity to learn new things, the way he could while he was at IIT Kharagpur. This motivated him to embark on graduate studies in the US.

He joined the University of Wisconsin–Madison for masters in Fall 2013, where he met Prof. Thomas Ristenpart. He started working with Tom on building a new kind of password manager during his masters. The project kindled his interest in computer security, which pushed him to pursue PhD in computer security with Dr. Ristenpart. In the mean time, Dr. Ristenpart decided to move from UW–Madison to Cornell Tech in New York City (NYC). Cornell Tech was a new initiative from Cornell University and Jacobs Institute to build a campus that will encourage more collaboration between academic research and industry. Cornell Tech offered only masters program. Therefore, Rahul decided to move to Cornell, Ithaca for PhD. After spending a semester in gorgeous Ithaca, Rahul moved to Cornell Tech in NYC, where he spent the remaining three and half years of his PhD.

During the graduate studies, Rahul has interned with Microsoft Research Technologies and with Dropbox Inc. Rahul is moving back to Madison to join the University of Wisconsin–Madison as an Assistant Professor in Fall 2019.

This document is dedicated to my Ma, Baba, and Sayangku.

ACKNOWLEDGEMENTS

Prof. Thomas Ristenpart, my advisor, is the reason I attempted to do a PhD and reached to a position to write this dissertation. His constant motivation, advice, and feedback helped me achieve what I am now. I am deeply grateful to Tom for pushing me to redefine my limits, for patiently working with me, and for continuously supporting me during my studies at the University of Wisconsin–Madison and at Cornell University.

Besides my advisor, I would like to thank Prof. Nicola Dell and Prof. Ari Juels for their insightful comments and guidance for my research and beyond. I learned a lot working with Nicki, especially the importance of human aspects in digital technologies we build. I am also thankful to Nicki for mentoring me in my academic job search process. I am fortunate to be able to work with Ari throughout my graduate studies. Ari is one of my role model researchers for his novel ways to look at a problem and expressing them precisely in the fewest possible words. Working with Nicki and Ari during my PhD widened both my research and myself as a researcher from various perspectives.

I am thankful to my colleagues and friends at Cornell University and at Cornell Tech for their stimulating discussions and encouragements. I am thankful to Cornell Tech for providing me the platform to conduct my research and finish my graduate studies. I am also thankful to all my co-authors and other friends who directly or indirectly contributed to and shaped this research. The list is long and I am surely going miss many whose critical involvement made this dissertation possible. Nevertheless, I would like to specially thank Adam, Andreas, Anthony, Antonio, Bijeeta, Buddhika, Deepak, Diana, Ethan, Fabian, Fan, Gerald, Ian, Joanne, Julia, Kim, Lei, Liang, Longqi, Lorenz, Lucy, Maurice, Neta, Nirvan, Paul, Phil, Saikat, Sam H., Sam S., Shoban, Sujay, Tal, Tyler, Vibhore,

Xiao, Yiqing, Yuval, and others for making my PhD experience memorable.

Last but not the least, I would like to thank my family: my parents, my brother, and my partner-in-life for supporting me throughout my graduate studies and my life in general.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Authenticating users to a digital system (such as a mobile phone, a computer, or a remote server) is a fundamental challenge in computer science. There are several mechanisms for authenticating users. The most prevalent ones rely on passwords. A password is a secret string shared between a user and a digital system, most often chosen by the user during a process known as *registration*. At the time of *login*, the user furnishes the same password (along with some user identifier). The concerned digital system checks if the provided password matches precisely with the one used during registration, and if so, the user is allowed to log in. We refer to such systems as password-based authentication (PBA) systems. PBA systems allow users to authenticate based on "what they know", such as a secret alphanumeric string. Besides passwords, users can also authenticate based on "who they are" or "what they have". Biometrics, such as fingerprints (Touch ID [3]), face (e.g., Face ID [4]), or iris, fall in the former category. Smart cards, physical security tokens (e.g., YubiKey [5]), or OAuth [6] fall in the latter category, where a user proves their identity based on proving possession of another account or a device.

Despite these alternatives, passwords are by far the most widely deployed and used authentication mechanism. This is because none of the alternatives to passwords comprehensively outperform passwords. Biometric authentication techniques lack the ability to unlink a user's physical identity from their digital identity. Moreover, unlike passwords, such physiological features, once compromised, cannot be changed easily. Account ownership-based authentication requires an authenticated account to seed the process, which is often authenticated based on a password. Device-based authentication requires additional hardware support.

Due to such shortcomings of other authentication techniques,[1] passwords remain the de facto authentication on the web and elsewhere.

To understand the current state of passwords, we need to look into its past. Therefore, I begin with a brief history of passwords and PBA systems, before delving into the core issues they face now.

1.1 A Brief History of Passwords

Finding an accurate history of passwords is very challenging, particularly because there is no clear definition of what is considered as a password and what is not. For example, Roman soldiers used secret *watchwords* to authenticate each other in the second century B.C. Watchwords were human-memorized secrets used for authenticating one soldier to another.

Even if we ignore such ancient use of "passwords", computer passwords have also been around for a relatively long time. The first computer password was used with MIT's Compatible Time-Sharing System (CTSS) in 1960 [8]. Fernando Corbató, one of the key developers of CTSS, designed the first password-based authentication system and integrated it with CTSS' multi-user environment, where passwords were used to ensure that users of CTSS mainframe system can keep their personal files private from each other [9]. In the early days, passwords of all users were stored in plaintext format in a single file, readable only by the system administrators. The file, however, got accidentally exposed to all users due to a software bug. Robert Morris and Ken Thompson recall this incident in their seminal paper on password security [10]:

[1] See [7] for more discussions on replacing passwords.

"Perhaps the most memorable such occasion occurred in the early 60s at a time when one of the authors (Morris) happened to be using the system. A system administrator on the CTSS system at MIT was editing the password file and another system administrator was editing the daily message that is printed on everyone's terminal on login. Due to a software design error, the temporary editor files of the two users were interchanged and thus, for a time, the password file was printed on every terminal when it was logged in."

This initial design of passwords was imperfect. Nevertheless, they became the de facto method for authenticating users to computers. Given the sensitivity of passwords, Robert Morris designed the first password hashing scheme using the cipher M-209 [11], used during World War II. Morris, who was a developer of the Multics system, included his hash function as a utility program named `crypt` (which was later inherited by Unix, and then by Linux). To make the hashes difficult to invert, he used the password as the key to the cipher (instead of the message) to encrypt a constant value. In the 1970s, he also introduced a standard file, called `/etc/passwd`, in Multics where password hashes were stored — which was also inherited by Unix systems and is still used. As the passwords were hashed and the password file also contained access control information necessary for many user-level programs, the `/etc/passwd` file was allowed to be read by anyone. But soon it was discovered that the hashing alone was utterly insufficient in protecting passwords: users pick very simple and easy-to-guess passwords (such as a single character or often use their username as the password). Therefore, an attacker, after obtaining the password file, can make repeated guesses for the user password and simulate login (offline) to discover user passwords from the hashes stored in `/etc/passwd`. This attack is known as a brute-force guessing attack and it can

be very damaging against human chosen passwords. Morris and Thompson showed that an attacker could recover 82% of user accounts in MIT's CTSS mainframe within a week worth of 1979's computing time [10].

Two important takeaways from the Morris-Thompson experiment are as follows. First, hashing passwords is not enough: proper access control of the password file is necessary for a multi-user system. Hence, password shadowing — the `/etc/shadow` file — was introduced in the mid 1980s along with the development of SunOS.[2] Second, password hashes, once leaked, can be inverted via brute-force guessing attacks; so we need ways to slow down checking a guess for a password hash. To do so, in the late 1990s, password-based key derivation functions (PBKDF) were introduced, which used iterated hashing to increase the cost of a single password hash. Though iterated hashing with off-the-shelf hash function, such as MD or SHA hashes, slows an attacker who is using a standard CPU to mount a guessing attack, its effect is negligible for an attacker with graphics processing units (GPUs) or application-specific integrated circuits (ASICs).

Ideally, the attacker must perform an equal amount of work for checking a guess as an honest server would have to. Servers typically use CPUs, but there is no limit on what hardware an attacker can use. This disparity in computation power between servers and attackers motivated research on different types of slow hash functions. In 1999, bcrypt [12] was designed. Bcrypt, in addition to using longer computation time, also requires a larger (4KB) amount of random memory accesses (compared to SHA or MD family hashes), thereby avoiding GPU or ASIC-based acceleration of password hashing. However, it turned out that a sophisticated FPGA with RAM blocks can expedite bcrypt computation significantly. To overcome this shortcoming, in 2009 Perchival introduced sequential memory-hard hash

[2] An implementation of the Unix operating system by Sun Microsystems.

functions and designed a special hash function called scrypt [13].

In the mean time, with the advent of the World Wide Web and public commercial use of the Internet in the late 1990s and its global proliferation in the next decade, passwords were being used to authenticate billions of people across the globe. PBA was the only feasible authentication mechanism that could support an internet scale variability of users and devices. Nevertheless, passwords were (and are) not without shortcomings. In particular, users tend to pick predictable passwords and reuse them across different websites. Websites are particularly bad at managing password databases. In the last decade, billions of passwords were compromised [14, 15] and posted online. Therefore, several attempts have been made to do away with passwords [16] and alternative techniques were proposed [17, 18]; nevertheless, none have replaced passwords entirely.

Interestingly (and ironically), leaked passwords provide a new perspective on how users choose passwords. While these leaks are often used to create better attacks against passwords, in this dissertation, I showed how to utilize these leaks to build better PBA systems.

Currently, an wide array of authentication techniques are being built to work in conjunction with passwords for different settings. To counter some of the security shortcomings of passwords, two-factor (2FA) or multi-factor (MFA) authentication techniques are being used, where the user not only provides a password, but also proves ownership of an account or a device. Moving forward, combinations of different authentication methods are expected to be used; which also means passwords will continue to be in use for the foreseeable future. Therefore, instead of despising passwords, we should invest in improving their usability and security. What follows is a discussion of some of the key issues with current passwords and

how this dissertation addresses them.

1.2 Password's Problems and their Solutions

One of the key design limitation of passwords is that the usability and security goals of passwords are inversely correlated: passwords must be easy to remember and type, while still be hard for an attacker to guess; and a user should pick completely different passwords for different websites and remember them, despite having hundreds of accounts; a user must enter the password accurately every time she wants to log in. Such conflicting goals[3] force users to *choose* between security and usability — obtaining effective security with passwords, in current settings, requires significant degradation in usability. Therefore, users make decisions that alleviate the usability burdens, often at the cost of security. The insecurity of passwords arises primarily due to its usability problems. Users constantly find ways to circumvent "the best advice" for handling passwords and, instead, make passwords the weakest link in computer security. Therefore, to improve the security of passwords, we need to address not only the security issues but also the usability issues associated with them.

I identify three key problems with passwords and current PBA systems that are impeding the usability of passwords and thereby degrading its security.

Too many passwords. Users nowadays have tens or even hundreds of accounts [22, 23]. Remembering different passwords for each of them is too cum-

[3]Not to mention, the tyranny of *password policies* that has been annoying users for more than a decade with outrageous requirements for creating a password. Such policies provide very limited security benefits [19, 20] at the expense of huge usability burdens. Though NIST finally do away with password policies in 2017 [21], it might take a decade or more for them to retire from the Internet completely.

bersome. To deal with the complexity of too many passwords, users pick easy-to-remember, simple passwords and reuse them across websites. All of which drastically degrade the security of passwords.

Experts advise using a password vault (also called password manager), where a user can store all of her account credentials — usernames and passwords — in one place and encrypt them using a master password. The user only needs to remember and enter the master password. Because password vaults hold all of a user's passwords, they are a lucrative target for attackers. Therefore, their security is extremely critical. Current password vaults however provide limited security once stolen. An attacker can try to recover the passwords stored in an encrypted password vault by brute-force decrypting the vault using guesses of the master password. Such an attack succeeds because traditional encryption mechanisms reveal clearly when the correct master password (or key) is used to decrypt the vault ciphertext. The attack is also *offline* in the sense that the attacker is not required to make any online queries. Therefore, the attacker is only bounded by the computational budget she can spend on decrypting the vault ciphertext. With the improved hardware (e.g., GPU cluster[4]) and powerful password guessing techniques (e.g., [24]), offline brute-force guessing attacks can be very damaging.

I therefore designed a new kind of password vault, called NoCrack, which uses honey encryption [25] to encrypt passwords under a master password. Decrypting a NoCrack vault ciphertext with an incorrect master password does not fail or output garbage values, rather it outputs a decoy but plausible looking set of passwords for the user's different accounts. A brute-force decryption attempt to recover the master password and the stored passwords from a NoCrack vault ciphertext

[4] https://arstechnica.com/information-technology/2012/12/25-gpu-cluster-cracks-every-standard-windows-password-in-6-hours/

results in millions or billions of decoy but plausible looking password vaults, one for each guess of the master password. The attacker is now faced with the challenge of finding out which of the decrypted vaults are real. The decrypted passwords has to be checked online on the websites the user has accounts with. Thus, the attack, which an attacker could pull off completely offline previously, now with NoCrack, requires online communications. Websites can monitor repeated login failures against an account and can flag anomalies in the password submissions. They can also lock the user's account to foil an attacker's further verification attempts, and thereby significantly restricting brute-force guessing attacks on the stolen password vaults.

Difficult to type. While password managers alleviate the burden of remembering and typing hundreds of user passwords, users still have to type the master password. A master password should be long and complex. Users often make mistakes while typing their passwords for decrypting password vaults or logging into an web account. In current settings, the decryption (or the login) attempt is rejected if the password is mistyped, and the user has to retype her password. This is very annoying and counter-productive for legitimate users. As shown in Appendix B.3, users take more time to type long and complex passwords and make more mistakes while typing them. Also, users have to type master passwords fairly often (e.g., LastPass[5] typically requires users to type their master passwords every time the browser session is restarted and after 8 hours of inactivity). Therefore, typos in such passwords can cause a significant usability burden on users, even when using a password vault.

I first quantified the problem of password typos. I partnered with Dropbox, a

[5]https://www.lastpass.com/

software company with hundreds of millions of users to instrument its production login infrastructure for 24 hours and measured how often users mistype their passwords. I found that 3% of Dropbox users fail to log in to Dropbox due to some simple, easy-to-correct typos such as accidentally capitalizing the first letter of the password or leaving the caps lock on; many more were delayed in their login due to these typos. Allowing users to log in with these small mistakes would significantly improve the usability of passwords. However, allowing user to log in with typos might degrade the security — the attacker can compromise a user account by guessing either her password or a typo of it. In this dissertation, I show allowing a specific subset of easy-to-correct typos does not degrade the security of user passwords relative to existing PBA systems, while still providing significant usability improvements.

While allowing some typos might be safe, it was unclear how to do that, given passwords are stored as cryptographic hashes: how to differentiate between genuine typos from an attacker's guesses. I proposed a corrector-based typo correction scheme, which can correct a small set of population wide popular typos by applying *corrector functions* to the incorrect password submission. Corrector functions are specifically designed functions for correcting particular types of typos. About 20% of password typos can be corrected using the corrector-based approach. However, this approach leaves 80% of users without the benefits of typo-tolerance.

I designed a new personalized password typo correction scheme, called TypTop, which monitors a user's password typing mistakes, and if a user makes the same mistake she made in the past, TypTop lets the user log in, should it be safe to do so. I formally analyzed the security of TypTop and proved that the security of user passwords remains unaffected due to typo-tolerance for real-world password

and typo distributions. I built TypTop as a Unix tool which renders a laptop login prompt typo-tolerant. Via a pilot deployment study with 24 users for a median of four weeks, I showed that TypTop can correct 63% of all password typing mistakes that users made, which is 300% more effective than corrector-based approaches.

Password reuse and data breaches. About 40% of users reuse their passwords on different websites [26, 27]. The difficulty of remembering too many passwords might encourage users to reuse their passwords. Nevertheless, password reuse leads to credential stuffing attacks, where an attacker tries to compromise a user account by simply using the leaked passwords of that user from other websites.

In the last decade, billions of user credentials — usernames and passwords — have been breached. The breached data is often posted online or sold on the Internet. Much of the leaked data, contained plaintext passwords (meaning the concerned websites did not bother to hash the passwords), while others were hashed with a fast hash function (e.g., MD5 or SHA1), or used no salt[6]. As a result, user passwords were recovered relatively quickly using brute-force guessing attacks.

With the availability of a daunting number of user passwords, credential stuffing attacks have become one of the most prevalent (and successful) vectors for account compromise [28]. Though password vaults would solve the problem of reusing passwords, the low adoption rate of password vaults [26] suggests we need complementary approaches to protect user accounts.

To prevent against credential stuffing attacks, many web services proactively look for leaked passwords and take action to protect accounts vulnerable to credential stuffing. Third-party services, such as HIBP [1] or Google's Password

[6] A salt is a per-user random string used to make hashes of different users unique, despite the underlying passwords being the same.

Checkup [2] provide APIs for checking leaked passwords, to which other web services can subscribe to know about whether or not a user password is leaked. We call such third-party services *compromised credential checking* (C3) services.

Such services often require the user to share a small prefix of the hash of the password with the C3 server. In this dissertation, I provided a thorough security analysis of the existing C3 protocols, and showed they can be very damaging to the secrecy of user passwords should the C3 server be compromised. An attacker can compromise 34% of user accounts in less than 10 guesses if the attacker obtains the prefix of the hash of those account passwords shared with the C3 server. This is 12x more than what an attacker could compromise without the hash prefix (2.8%). The security loss is even higher if the user has been affected by a prior data breach. I went on to give more secure protocols for checking compromised credentials that provide better security/bandwidth trade-off.

1.3 Methodology: Empiricism-Informed System Design

The main challenges in designing a real-world system — particularly ones that include human actors, such as PBA systems — lie in understanding the details of how users and attackers are going to interact with the system. Often the lack of understanding leads to insecure or unusable deigns. For example, the use of complicated password policies and periodic forced password reset are two such design choices that provide minimal security benefits at the cost of huge usability degradation.

Traditionally, security of a system is argued using analytical techniques. However, analytical techniques alone fail to identify all the details of real-world systems,

and are therefore ineffective for analyzing the security and usability of them. Similarly, purely data-driven approaches are also insufficient to rigorously argue the security of a system. Conclusions drawn based on empirical analysis is subject to change based on the variability of the underlying data.

I therefore took a different approach that combines empirical techniques with analytical methods to design and evaluate PBA systems. I used analytical tools to understand security of a system under well-defined assumptions, and then verified them using empirical methods. I followed an iterative four-step approach to design and build new systems. In each part of this dissertation, I began with *measurements* to understand a specific part of the problem, then used that knowledge to guide *designing* new solutions. I then *evaluated* the performance and the security of those solutions empirically and analytically. Finally, I *built* a system to test how the solutions fit in the larger landscape, and iterate. Such a measure-design-evaluate-build-measure cycle is what I refer to as empiricism-informed system designing.

A key contribution of this dissertation is to put empiricism-informed system design into a solid, explicit framework for building real-world secure systems. I showed this approach can be very useful to design and build new solutions for passwords and PBA systems. I believe this generic framework can also be applied to designing and building real-world secure systems.

1.4 Contribution and Outline

The security of passwords is deeply intertwined with its usability. Often this connection is either overlooked or ignored because of the belief that usable passwords

are unlikely to be secure. One of the key contribution of this dissertation is to show that it is possible to improve usability of passwords without compromising its security. Using a combination of analytical and empirical methods I designed and built several solutions to overcome some of the topical problems with passwords.

The rest of the dissertation is organized in four chapters covering works from [29, 30, 31, 32] done in collaboration with Devdatta Akhawe (Dropbox), Junade Ali (Cloudflare), Anish Athalye (MIT), Joseph Bonneau (NYU), Anusha Chowdhury (Cornell), Ari Juels (Cornell Tech), Lucy Li (Cornell Tech), Yuval Pnueli (Technion), Bijeeta Pal (Cornell Tech), Thomas Ristenpart (Cornell Tech), Nick Sullivan (Cloudflare) and Joanne Woodage (Royal Holloway).

- In Chapter 2, I show how to design a novel decoy based password vault system, called NoCrack, that can resist offline brute-force attack.
- In Chapter 3, I show, via studies on Amazon Mechanical Turk and with Dropbox production authentication service, that password typos cause a significant usability burden. I demonstrate how to correct a carefully chosen set of typos that will improve usability without any degradation in security.
- The typo correction can be further improved by personalizing the corrections. In Chapter 4, I design a system called TypTop, which, once engaged with a authentication service, learns a user's password typing mistakes over time, and let the user login with frequent mistakes that are safe to do so. I show TypTop can correct 66% of all typos that users make. TypTop provides equal protection against untargeted attacks as traditional authentication systems do.
- In Chapter 5, I create a framework to analyze the security and computational overhead of protocols for checking against compromised passwords hosted by

third party services. I show that the protocols currently in use can severely damage password security should an attacker compromise the server providing leaked password checking service. I go on to give more secure protocols for checking compromised passwords hosted by third party services, that are comparable in computational overhead.

CHAPTER 2

CRACKING-RESISTANT PASSWORD VAULTS

This chapter was published in IEEE Symposium on Security and Privacy (S&P), 2015 [29].

2.1 Introduction

To alleviate the burden of memorization, many security experts recommend the use of *password vaults*, also known as "wallets" or "managers" [7]. Password vaults are applications that store a user's website passwords, along with the associated domains, in a small database that may be encrypted under a single *master password*. Vaults enable users to select stronger passwords for each website (since they need not memorize them) or, better yet, use cryptographically strong, randomly chosen keys as passwords. Many modern vault applications also backup encrypted vaults with an online storage service, ensuring that as long as users remember their master passwords, they can always recover their stored credentials.

These vault management services provide attackers with a rich target for compromise. The popular password vault service LastPass, for example, reported a suspected breach in 2011 [33]. Security analyses of several popular web-based password managers revealed critical vulnerabilities in all of them, including bugs which allow a malicious web site to exfiltrate the encrypted password vault from the browser [34].

An attacker that captures users' encrypted vaults can mount offline brute-force attacks against them, attempting decryption using (what the attacker believes to be) the most probable master password mpw_1, the second most probable mpw_2,

and so on. With standard password-based encryption (PBE) algorithms (e.g., PKCS#5 [35]), only the correct master password produces a valid decryption, so an attacker knows when a vault has been successfully cracked. Such attacks are offline in the sense that they can be conducted without interaction with a server and hence are limited only by the attacker's computational capabilities, as measured by the number of decryption attempts performed per second.

Available evidence (c.f., [36]) suggests that most master passwords selected by users are weak in the sense that even a modestly resourced attacker can feasibly crack them in a matter of minutes or hours.

This state of affairs prompts the following core question addressed by our work: *Can password vaults be encrypted under low-entropy passwords, yet resist offline attacks?* We call such vaults *cracking-resistant*. Cracking-resistant vaults would have the benefit that attackers, even after investing significant resources in offline cracking, would still need to perform a large number of online login attempts. Such online login attempts would significantly impede attackers and enable service providers to detect vault compromises evidenced by failed login attempts.

Breaking Kamouflage. At first glance, it may seem that preventing offline cracking is impossible. After all, the attacker has access to the ciphertext and the key is likely low-entropy and so easily guessable. But Bojinov, Bursztein, Boyen, and Boneh, who first investigated the question of cracking-resistance for vaults, suggest a clever idea: include decoy or honey vaults in an attempt to force offline attacks to become online attacks [37]. Their system, called Kamouflage, permits seemingly successful decryption under some *incorrect* master passwords. Kamouflage stores not only an encryption of the true vault (under the true master password) but also a large number $N - 1$ of decoy vaults encrypted under decoy

master passwords. The security goal is for an offline attack to reveal only a set of equally plausible vaults, forcing an attacker to attempt a login with a password from each of the decrypted vaults in order to find the true one. The quantitative security claimed in [37] is that an attacker must first perform offline work equivalent to that needed to crack conventional PBE, but additionally perform an expected $N/2$ online queries.

Our first contribution is discovery of a subtle vulnerability in this approach; we show a cracking attack that exploits it, and evaluate the attack's efficacy. To explain briefly, Kamouflage stores $N - 1$ decoy vaults encrypted under decoy master passwords that are generated by parsing the true master password and using its structure to help select the decoys. Analysis of password leaks shows that learning the true master password's structure reduces the attacker's search space substantially more than one might expect. Successfully cracking *any* of the N vaults (decoy or true), therefore, reveals this structure and confers a big speed-up on an attacker. We explain the issue in greater detail in the body.

The upshot is that using Kamouflage actually *degrades* overall security relative to traditional PBE: an attacker can perform significantly less computational work (on average about 40% less for N = 1,000) and make just a handful of online queries. In practice, an attack against Kamouflage would therefore require less total computational resource than one against traditional PBE, in almost all cases.

Unfortunately this flaw is not easy to fix, but rather seems fundamental to the Kamouflage design. Thus our finding re-opens the question of how to build cracking-resistant vaults.

Natural language encoders. Our next contribution is a new approach to

cracking-resistant vault design. The vulnerability in Kamouflage is leakage of information by the explicitly stored set of decoy ciphertexts about the true master password. So we seek a design that stores a single explicit ciphertext and has the property that decryption with the wrong key ends up generating on-the-fly decoy vaults. Here we are inspired by Hoover and Kausik's cryptographic camouflage system [38] and a generalization called honey encryption (HE) [25]. These prior schemes, however, work on plaintexts that are drawn from simple distributions, i.e., uniform over some set. They would work (with some adaptations) for computer-generated random passwords, but fail, as we confirm experimentally, to impart cracking-resistance to password managers that include human-selected passwords.

To meet the challenge of good decoy vault generation, we introduce the concept of *natural language encoders* (NLEs). An NLE is an algorithm that encodes natural language texts (in our case, lists of human-selected passwords) such that decoding a uniformly selected bit string results in a fresh sample of natural language text. Technically, NLEs are a new class of the more general concept of a distribution-transforming encoder [25], previous examples of which generated uniform prime numbers and uniform integers. NLEs require different techniques given the (widely understood) additional complexity of modeling natural languages, even in the restricted domain of passwords [39, 24, 40].

Rather than design NLEs from scratch, we instead show how to convert existing models from statistical natural language processing into NLEs. Models are compact representations of the distribution of natural language texts; particular choices might include n-gram Markov models, probabilistic context-free grammars (PCFGs), etc. We focus on the first two because they have been used to build effective password crackers [39, 24, 40, 41]. Our constructions of NLEs from these

model types therefore gives, in effect, a way to retool password crackers to help us build cracking-resistant vaults. This approach has significant benefits, as it permits future improvements to natural language models to be incorporated easily into password vault systems.

We build a number of NLEs based on existing models trained from password leaks as well as one new PCFG-based model. To evaluate security, we must determine whether decoys generated via the decode function are distinguishable from true passwords. More specifically, given a sample that is either a decoy or a random sample from a password leak (which acts as proxy for a real password), the adversary must classify its input as a decoy or not. We experimentally evaluate a number of machine learning classification attacks that allow the adversary to train on decoys as well as passwords from the true distribution. These analyses show that basic machine-learning attacks do not break our schemes. A human might be able to distinguish real from decoy, but forcing attackers to rely on the manual effort of humans (across thousands of decoys in a typical parametrization) would also be a huge improvement in security over existing solutions.

We also provide the first model of the password sets associated with password vaults which may include related or repeated passwords. Unlike single-password models, for which there is a large amount of public data available, there is not yet much public data available to researchers on full password vaults. (This dearth of data is also presumably problematic for would-be attackers.) Again we perform a preliminary analysis using a small number of leaked vaults. This analysis suggests that an extension of our PCFG-based NLEs generates password vaults resistant to simple machine-learning attacks that attempt to distinguish fake vaults from real ones. We caution that future work might find better attacks against our particular

NLEs. By construction, however, our approach never yields security worse than conventional PBE, unlike the case with Kamouflage (as shown by our attack). Our NLEs at least provide another obstacle that crackers must find ways to overcome, one that may be bolstered by improvements in language models.

A full vault system. We incorporate NLEs into a full encrypted vault service called NoCrack that resists offline brute-force attacks better than alternative password vault schemes. NoCrack addresses several previously unexplored challenges that arise in practical deployment of a honey vault system: concealing website names associated with vault passwords, incorporating computer-generated passwords in addition to user-selected ones, and authenticating users to the NoCrack service while ensuring service compromise does not permit offline brute-force attacks. We report on a prototype implementation of NoCrack that will be made public and open-source.

Summary. In summary, our contributions include:

- *Breaking Kamouflage:* We show that Kamouflage provides less security than its original analysis suggested, and in most realistic cases even less than traditional PBE schemes.
- *Natural-language encoders (NLEs) for vaults:* We introduce the concept of natural language encoders, and show how to build them from typical password models. This approach allows us to generate realistic decoys on the fly during brute-force attacks.
- *Cracking-resistant password vaults:* We use our new NLEs as the basis for a password vault system called NoCrack. It addresses a number of issues for the first time, including how to deal with authentication, concealment of the websites

for which a user stores vault passwords, and more.

2.2 Background and Existing Approaches

In practice, password vaults are encrypted under a user's master password using a password based encryption (PBE) scheme. In more detail, a user's set $\vec{w}$ of passwords is encrypted under a master password mpw using authenticated encryption. Encryption and decryption use a password-based key-derivation function (KDF), meaning encryption is under a key computed by applying a hash chain to a random salt and mpw [35]. (The salt is stored with the ciphertext.) A decryption operation under an incorrect master password mpw' will fail, as it will not authenticate correctly[1]. It is for this reason that an adversary can mount an offline brute-force attack against a vault. The adversary's trial decryption will fail until the correct master password mpw is discovered.

Currently in-use KDFs prevent precomputation attacks [42] (such as rainbow tables [43]) by salting. They also increase the cost of brute-force attacks by iterated hashing, which slows down each decryption attempt. See [44] for a formal analysis. But all this only slows down brute-force attacks and does not prevent them; the protection conferred on $\vec{w}$ by PBE is ultimately a function of the resistance of mpw to guessing attacks. Numerous studies, e.g., [45], have shown that users tend in general to select passwords that have low guessing entropy. There is no available evidence indicating that users choose significantly stronger master passwords. Given the growing use of password vaults by consumers and the vulnerabilities they introduce — bulk compromise on servers or compromise on consumer devices

[1]Some systems may use unauthenticated encryption such as CBC mode. Decryption under the wrong master password is still detectable with overwhelming probability, c.f. [25].

— the risk of brute-force attacks is real and pressing.

Enter decoys. It is the inherent limitations of conventional PBE in thwarting offline brute-force attack that motivated Bojinov et al. [37] to propose the Kamouflage system. As explained above, the idea behind Kamouflage is to enable multiple master passwords to successfully decrypt a vault, while only the true master password yields the correct vault plaintext. The hope is that the attacker cannot distinguish between real and decoy offline, and must instead make online queries using decrypted credentials to identify the true vault.

Specifically, to protect the true master password and vault $(mpw^*, \vec{w}^*)$, Kamouflage generates $N-1$ decoy $(mpw_i, \vec{w}_i)$ pairs and stores them in a list. Decoys are generated using dictionaries of commonly seen tokens (strings of contiguous letters, numbers or symbols) found in a password leak. We describe decoy generation in more detail in the next section. Every $\vec{w}_i$ is encrypted using a conventional PBE scheme under mpw_i and kept at a location L_j that also calculated from mpw_i. If a collision results during the generation process, i.e., a master password is created that maps to an already occupied location, then a fresh $(mpw_i, \vec{w}_i)$ pair is generated. Thus, given master password mpw', it is possible to locate the corresponding vault and attempt to decrypt it with mpw'.

Kamouflage has some notable deployment limitations. As it stores N vaults, its storage cost is linear in the security parameter N. Additionally, Kamouflage discourages user passwords that are not parsable using its dictionaries, a major obstacle to practical use. Such password rejection may also degrade security by encouraging users to choose weaker passwords than they otherwise might. Finally, Kamouflage provides no guidance on protecting the confidentiality of domains paired with the passwords in a vault and on storing computer-generated passwords,

Statistics	RockYou	RY-tr	RY-ts	Myspace	Yahoo
Number of accounts	32.6 M	29.6 M	2.98 M	41,537	442,846
Number of Unique passwords	14.3 M	13.0 M	1.3 M	37,136	342,517
Min-entropy (bits)	6.8	6.7	5.3	9.1	8.1
Avg. password length	7.9	7.9	7.9	8.5	8.3
Avg. letters per password	5.7	5.7	5.6	6.4	6.2
Avg. digits per password	2.2	2.1	2.3	1.8	2.0
Avg. symbols per password	0.05	0.05	0.05	0.3	0.04
Letter-only passwords	44.0%	44.1%	43.8%	7.0%	34.6%
Passwords w/ digits	52.2%	54.0%	54.3%	84.8%	64.7%
Passwords w/ symbols	3.7%	3.7%	3.7%	10.7%	2.8%

Figure 2.1: Statistics of the password leak datasets used in this chapter.

both features of today's commercial systems. More fundamentally, the approach of hiding the true vault among an explicitly stored list of decoy vaults has inherent security limitations, as we explain in the next section.

Threat Model. The primary threat model that we consider is theft of encrypted password vault due to adversarial compromise of a vault storage service, a lost or stolen client device, or a software vulnerability allowing exfiltration of the encrypted vault (c.f., [34]). The attacker does not know anything about the passwords present inside the vault but she has knowledge about the distribution of human generated passwords (learned from publicly available password leaks). She also knows the encryption algorithm and the other information (if any) used by the algorithm at the time of encryption or decryption. She does not know the master password or any randomness consumed by the encryption algorithm.

Her objective is to learn the correct master password (and corresponding in-vault passwords) using a minimal amount of computation and online querying. She tries to decrypt the vault using offline brute force attack against the master password. She tests the correctness of a vault by attempting to log into a domain with a corresponding credential in the vault.

We will measure both the offline and online resources used by the attacker in terms of number of decryption attempts and number of online queries, respectively. To attack a conventionally encrypted vault, no online queries are required and the offline work is a function of the adversary's uncertainty about the master password; for Kamouflage, the claimed security is offline work equivalent to that of conventional encryption and an expected $N/2$ online queries. In each of our experiments, we will calculate expected offline and online work over a choice of master password drawn from a distribution that we will explicitly specify. The distributions we use will be informed by password leaks, as we now discuss.

Datasets. In the course of this chapter, we use a number of datasets to train password language models, to train attackers, and to test attackers. We primarily use three large-scale password leaks: RockYou, Myspace, and Yahoo. RockYou is the largest leaked clear-text password set to date and is used extensively in modeling distributions of human-chosen passwords [39, 40, 46]. The Myspace leak occurred when passwords for user accounts were publicly posted; the passwords were gleaned from a phishing attack against Myspace's home page. In 2012, Yahoo lost nearly 450,000 passwords after a server breach. Note that all of these leaks contain only original plaintext passwords and not cracked password hashes, as in some other leaks.

As we wish to ensure different training sets for the password model and attacker in some cases, we partition the RockYou passwords into two sets, randomly assigning 90% of the passwords to a set denoted RY-tr and the remaining 10% to RY-(ts). We train our language model with RY-tr only, but use all sets for adversarial training and testing (with cross-fold validation as appropriate). Figure 2.1 presents statistics on our data sets. Given its large size, we did not make use of

multiple splits of the RockYou data set in our experiments. We use RY-ts as a testing set to model settings in which the adversary and defender have equivalently accurate knowledge of the distribution from which user passwords originate. The Myspace and Yahoo data sets serve to model more challenging scenarios in which the adversary has more accurate knowledge of the password distribution than the defender.

2.3 Cracking Kamouflage

The stated security goal for Kamouflage is that given an encrypted vault, an attacker must first perform an offline brute-force attack to recover all of the plaintext vaults and then perform an expected $N/2$ online login attempts to identify the true vault. Through simulations using the leaked password datasets discussed above, we now show that Kamouflage falls short of its intended security goal. We first present more details on Kamouflage, and then describe and analyze an attack against it.

Decoy generation in Kamouflage. To construct decoy plaintext vaults, Kamouflage uses an approach based on deriving templates from the true plaintext vault. Let $\texttt{L}_n$, $\texttt{D}_n$, and $\texttt{S}_n$ each be a subset of all n-digit strings containing only upper or lower case English letters, only decimal digits, and only punctuation marks and other common ASCII symbols, respectively. We refer to the sets $\texttt{L}_n$, $\texttt{D}_n$, and $\texttt{S}_n$ as *token dictionaries* and individual strings in token dictionaries as *tokens*. No token appears in multiple token dictionaries. Token dictionaries are initially populated with tokens by parsing the passwords in a password leak[2] and placing each resulting token in the appropriate dictionary. Here parsing of each password in the leak

[2]Also suggested in [37] is to use the dictionaries used by password cracking tools such as "John the Ripper". In our experience leaks work better for Kamouflage.

is performed by greedily picking the longest contiguous prefix of just letters, digits, or symbols of a password, moving this prefix to the appropriate token dictionary, and then repeating the process on the remainder of the original password.

Once token dictionaries are fixed, Kamouflage parses a user-input password as a sequence of tokens found in the dictionaries. Successful parsing yields a sequence of tokens whose concatenation equals the original password. As above, parsing involves greedy decomposition of the password into tokens of contiguous letter, digit, or symbol strings; the presence of each token in the appropriate dictionary is then checked. (If any token is not present in its appropriate dictionary, then Kamouflage rejects the input password and prompts the user to pick another.)

A *password template* is the structural description of a password, expressed as the sequence of token dictionaries corresponding to tokens yielded by parsing of the password. For example, the password `password@123` is parsed as `password`, `@`, `123`, and the associated template is therefore $\mathtt{L}_8\mathtt{S}_1\mathtt{D}_3$.

A *vault template* extends the notion of a password template to sequences of passwords, while also keeping track of reuse of the same token in multiple locations. For a sequence of passwords $mpw^*, \vec{w}^*$, the vault template is generated as follows. First, parse each password. Then, for each unique token across all of the parsings, replace it with a symbol $\mathtt{X}_n^i$ for $\mathtt{X} \in \{\mathtt{L}, \mathtt{D}, \mathtt{S}\}$ where n denotes the length of the token and i denotes that this is the i^{th} token in $\mathtt{X}_n$ found in the sequence of parsings. The resulting sequence of password templates constitutes the vault template. In the example below, the first column contains the passwords composing a small vault: a master password followed by two in-vault passwords (for logging into websites). The second column contains the sequence of corresponding password templates that make up the full vault template.

`password@456123`		$\mathtt{L}_8^1\mathtt{S}_1^1\mathtt{D}_6^1$
`password4site`	$\rightarrow$	$\mathtt{L}_8^1\mathtt{D}_1^1\mathtt{L}_4^1$
`bob!Site`		$\mathtt{L}_3^1\mathtt{S}_1^2\mathtt{L}_4^2$

Observe that the symbol $\mathtt{L}_8^1$ is used twice, as the substring `password` appears twice, but the distinct substrings `site` and `Site` are respectively replaced by distinct symbols $\mathtt{L}_4^1$ and $\mathtt{L}_4^2$.

Given the vault template for $(mpw^*, \vec{w}^*)$, Kamouflage produces each decoy vault $(mpw_i, \vec{w}_i)$ by replacing each unique symbol X_n^i with a token chosen uniformly at random from the token dictionary X_n.

The attack. We exploit two vulnerabilities in Kamouflage. First, all of the decoy master passwords have the same template as the true master password. As soon as an adversary recovers any (decoy or real) master password during an offline brute-force attack, the corresponding template is revealed. Knowledge of this template enables the adversary to narrow its search significantly, to master passwords that match the revealed template. This strategy permits an offline attack to be accelerated to the point where it is *faster than cracking a conventional PBE ciphertext.*

Second, decoy master passwords are chosen *uniformly* with respect to the master-password template, i.e., tokens are selected uniformly at random from their respective symbol dictionaries. The decoy master passwords that result are distributed differently than real, user-selected passwords. Thus, if an adversary guesses master passwords in order of popularity, the real master password is more likely to be assigned a high rank (and so guessed sooner) than the decoy master passwords.

Given these two vulnerabilities, we craft an attack that employs a simple model of password likelihood in which the probability of a password is the product of the probability of its template and the probabilities of replacements for each template symbol. For example, $\Pr[\texttt{password9}] = \Pr[\texttt{L}_8\texttt{D}_1] \cdot \Pr[\texttt{password}|\texttt{L}_8] \cdot \Pr[\texttt{9}|\texttt{D}_1]$. The model is trained using a password leak. Specifically, $\Pr[\texttt{L}_8\texttt{D}_1]$ is defined to be the empirical probability that a password in the dataset has template $\texttt{L}_8\texttt{D}_1$, and similarly for other passwords. (Other models may be used, of course.)

Given this model and a challenge consisting of N Kamouflage-encrypted vaults, the attack proceeds using two guessing strategies, one offline and one online. First, in an offline effort, the attacker generates trial master passwords in decreasing order of their probability within the password-likelihood model, until one decrypts one of the N vaults decrypts successfully. At this point, the adversary has learned the template of the true master password and may narrow its offline search to master passwords that match this template, still in order of their probability within the password-likelihood model.

Upon decrypting any vault successfully, i.e., discovering its corresponding (true or decoy) master password, the attacker makes an online login attempt against a website[3] with one of the retrieved vault passwords. If the login succeeds, the adversary has identified the true vault and halts. Otherwise, the adversary resumes the offline attack against master passwords.

Attack evaluation. To evaluate the speedup of our attack over a naïve Kamouflage-cracking strategy, we perform simulations in a simplified attack model. The adversary is given a Kamouflage-encrypted vault and access to an oracle that indicates whether a queried password is the true master password or not. (This

[3]Recall that Kamouflage does not encrypt the sites associated with each entry in the vault.

oracle corresponds in a real attack to an adversary's ability to test a password from a decrypted vault via an online query to a real website.) We count oracle queries as well as offline decryption attempts. (We treat the KDF-induced slowdown as a unit measure.) Thus the challenger takes a master password (mpw^*) and a number N, and generates an encrypted Kamouflage vault set of size N. Then the attacker is given this vault set and access to the oracle. Its goal is to guess mpw^*.

The Kamouflage decoy generation algorithm uses a dictionary of replacements for $\mathtt{X_n}$. Given that this is public (being used by any implementation of Kamouflage), we assume the adversary has access to it as well.

Evaluation. We use the RY-tr dataset for training both Kamouflage's parser (i.e., populating the token dictionaries) and the attacker mentioned above. We use master passwords sampled from the RY-ts, Myspace, and Yahoo leaks for testing the performance of the attack. Our evaluation therefore covers both the case when master passwords are chosen from a distribution (RY-ts) similar to that used to train Kamouflage as well as the case when they are chosen from a difference distribution (Myspace and Yahoo). For every password in each of the three sets, we use the password as the true master password mpw. We then construct $N-1$ decoy master passwords[4] for both $N = 10^3$ and $N = 10^4$ (the values suggested in [37]). We then calculate the median offline and online work of 100 iterations for each (mpw, N) combination using fresh coins for decoy generation in each iteration.

We used only the first 50 million trial master passwords generated by the attacker's password-likelihood model for the attack, meaning the attack will not succeed against every master password in the datasets. We also exclude any pass-

[4]Since our attack does not exploit the vault contents, we dispense with simulating generation of site passwords.

Approach	N	Myspace		Yahoo		RY-ts	
		Offline ($\times 10^3$)	Online	Offline ($\times 10^3$)	Online	Offline ($\times 10^3$)	Online
PBE	–	5,740	0	2,467	0	114	0
Kamouflage	10^3	2,550	11	1,261	22	84	2
	10^4	1,381	108	751	219	64	18
Kamouflage+	10^3	165	344	76	302	9	190
	10^4	150	3,118	70	2,168	7	777

Figure 2.2: The expected amount of offline work (rounded to nearest thousand decryption attempts for integer) and online work (rounded to nearest number of login attempts to a website) required to break a conventional vault (PBE), Kamouflage, and Kamouflage+ for $N \in \{10^3, 10^4\}$. Here expectations are taken over the distribution of Myspace, Yahoo, or RY-ts passwords, normalized after removing those that cannot be parsed using the Kamouflage grammar and that cannot be cracked using the first 50 million guesses of our cracker.

words that can't be parsed by Kamouflage (only 0.01%, 11.43%, and 13.49% of passwords for RY-ts, Myspace, and Yahoo).

This allows us to compute a number of statistics regarding the attack's efficacy. We start with the average difficulty of cracking a given password sampled from the three challenge distributions, assuming it is crackable using the first 50 million guesses by our password cracker. Figure 2.2 gives a breakdown of this statistic across the various settings. (The results for Kamouflage+ are explained below.)

The takeaway is that for $N = 10^3$, breaking a Kamouflage vault requires on average only 44% of the computational cost of breaking PBE and incurs only 11 online queries if master passwords are sampled from Myspace leak. If the value of N is increased to 10^4, an attacker needs less than 24% of the computational work required in the case of PBE to break a Kamouflage vault. The offline work therefore goes down with increasing N, which stems from the fact that more decoys means more chances of learning the master-password template early in the attack. The online work does increase with increasing N, for $N = 10^4$ requiring an expected

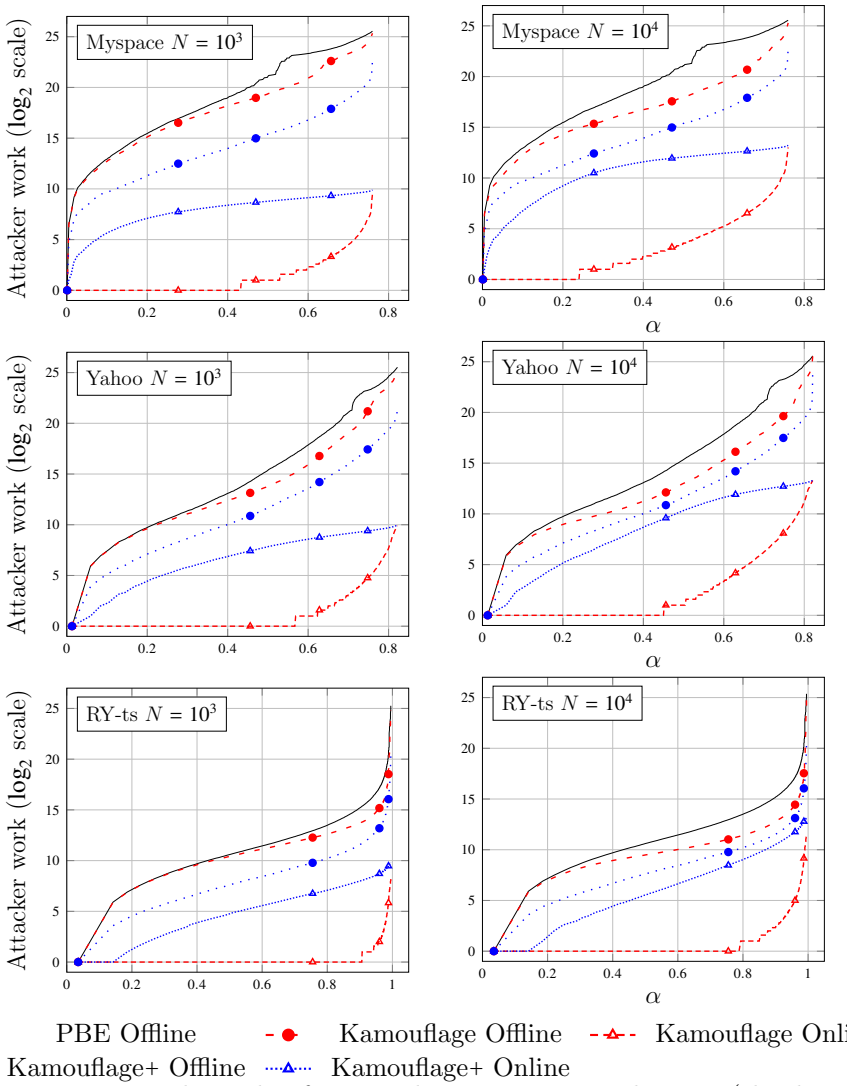


Figure 2.3: Experimental results for attacking conventional PBE (the baseline) and for Kamouflage and Kamouflage+ vaults for $N \in \{10^3, 10^4\}$. Shown is the α-guesswork measured in number of offline attacker decryptions and number of online attacker queries (both $\log_2$ scale).

108 queries for Myspace or 219 for Yahoo challenges. Nevertheless, all the numbers for Kamouflage are an order of magnitude smaller than the goal that an attacker must make expected number of queries $N/2 = 5,000$ (for $N = 10^4$). For RY-ts the offline speedup is less but the attacker requires only 2 online queries on an average for $N = 10^3$ and 18 for $N = 10^4$ to break RY-ts challenge. The PBE numbers are also quite low for RY-ts: the attacker does better when trained on a master password distribution close to that used by the target user.

A perhaps more refined measure of attack success is the α-guesswork factor introduced by Bonneau [45]. This factor measures the work required to break any αt-sized subset of a corpus of t total passwords. So at $\alpha = 0.5$, we give the α-guesswork in terms of the number of offline computations to break any half of the passwords in each leak and the number of online queries to break any half of the passwords. Figure 2.3 shows the results for the range of α we considered. The graphs end at $\alpha = 0.82$ for Yahoo and $\alpha = 0.76$ for Myspace, reflecting the 18% and 24% of challenge passwords we either did not crack in the first 50 million guesses or which were not parsable by Kamouflage. For RY-ts the first 50 million were able to crack all but a handful of the passwords. While we display both online and offline queries on the same charts, they do not necessarily correspond to the same subsets of master passwords.

The charts give a more expansive view of Kamouflage's resistance to attacks as compared to conventional PBE. Looking at the top left chart in Figure 2.3, we see that for conventional PBE (the highest solid black line), up to $\alpha = 0.5$ of Myspace passwords can be cracked in 2^{21} offline decryptions, while for Kamouflage the same can be achieved with only 2^{19} offline queries (second highest red dashed line). Half of the master passwords can be broken in two online queries for $N = 10^3$ (lowest densely dashed red line). Increasing Kamouflage's security parameter to $N = 10^4$ (bottom left chart) slightly increases the online work for some α (bottom blue curve), but at the cost of yielding even more efficient offline decryption costs. Breaking Kamouflage for a set of $\alpha = 0.5$ of Myspace passwords is achievable with 2^{15} offline computations. The Yahoo charts (middle two) show a similar picture. Half the master passwords can be cracked in just two online queries for $N = 10^3$. The attack performs best in the case of RY-ts. Here the attacker can break more than 95% of master passwords with only one online query (top right chart). This

is because the attacker has a very good estimate of the distribution of the user's true password.

The largest number of online queries needed at $\alpha = 0.5$ across all six settings is just 11 for Myspace with $N = 10^4$.

Kamouflage+. Our attack against Kamouflage required few online queries, which is due to the fact that decoys were mostly ranked lower by our cracker than the true master password. This vulnerability arises because Kamouflage's uniform selection of tokens produces relatively unlikely decoy master passwords. We therefore experimented with a modified version of Kamouflage, called Kamouflage+, that instead chooses tokens according to their frequency in the RY-tr dataset. We repeated simulations of our attack, exactly as above but now with decoy master passwords generated by Kamouflage+. The expected work for our various experimental configurations is given in Figure 2.2. The results in terms of α-guesswork are shown in Figure 2.3 (the dotted blue line, third from top being offline work and the dotted blue line second from bottom being online work).

This modification to Kamouflage actually backfires in the case of offline work. This is because decoys are now more likely to be probable passwords, speeding up even further the attacker's discovery of the master password template. On the other hand, Kamouflage+ does increase the online work for our attack, but this is still noticeably below the intended goal of $N/2$ in all cases. Even with $N = 10^4$, the expected number of online queries is never more than 65% of the goal of 5,000 queries.

Summary. This relatively simple attack shows how existing approaches to cracking-resistant vaults actually degrade security relative to conventional PBE.

Specifically, the general Kamouflage approach of storing an explicit list of ciphertexts, with one true vault embedded among many decoys generated as a function of the true master password, provides brute-force attackers with an offline speedup that increases with the number of decoys. Attackers may even be able to do better in the online portion should they analyze the vault contents more carefully (our attack above orders online queries solely by master password likelihood).

The lesson here is a warning against building decoys as a function of the true master password. One might consider modifying Kamouflage to use decoy master passwords independently sampled from a model of the password distribution, in order to eliminate the speedups obtained by our attack. But this approach gives rise to other limitations, such as a cap of $N/2$ expected online queries irrespective of the entropy of the true master password. We therefore go a different route.

2.4 Overview of Our Approach

The approach of hiding a true vault in a list of encrypted vaults appears to have fundamental limitations. Most obviously it only allows a number of decoys linear in the storage size, and the construction of effective decoy master passwords, as shown for Kamouflage in the last section, is challenging.

We take a significantly different approach. Instead of explicitly storing decoy vaults, we construct single ciphertext which, when decrypted with *any* wrong master password, yields a decoy vault that appears to have been sampled from the distribution of plaintext vaults (across the entire user population). This approach is inspired by the theory of honey encryption [25]. As we will see, though, successfully building a honey vault using honey encryption raises a number of chal-

lenges, chief among them accurately modeling the distribution of human-generated password sets in vaults.

Honey encryption. Juels and Ristenpart [25] introduced *honey encryption* (HE) as a mechanism for encryption of secrets under low-entropy passwords. We describe their general approach, with terminology tailored to our setting. Encryption with HE takes as input a master password mpw and plaintext M and outputs a ciphertext, which we denote by $C = \mathsf{HEnc}(mpw, M)$. Encryption is usually randomized. Decryption takes as input a master password and ciphertext C, and outputs a plaintext, denoted $M = \mathsf{HDec}(mpw, C)$. In our setting, a plaintext is a single password w or vector of passwords $\vec{w}$. We require that $\mathsf{HDec}(mpw, \mathsf{HEnc}(mpw, M)) = M$ with probability one for all mpw, M (over any randomness used in encryption).

HE schemes are designed so that a ciphertext, when decrypted under an incorrect master password $mpw' \neq mpw$, emits a "plausible" plaintext M'. This requires a good model, built into the HE scheme specification, of the distribution from which plaintexts are drawn.

As a concrete example Juels and Ristenpart gave an HE scheme for a message M that is a uniformly sampled prime number. (This scheme can be leveraged to encrypted RSA private keys.) Their construction follows a general approach to building HE schemes that composes a distribution-transforming encoder (DTE) with a carefully chosen, but still conventional, PBE scheme. The latter can be, for example, CTR-mode encryption with key derived from the master password using a PBKDF. A DTE scheme specifies a randomized encoding of a message as a string of bits. Decoding does the reverse (deterministically). For a secure HE scheme, it is a requirement that the output of the encoding, for M drawn from

some target distribution, looks uniformly distributed and that decoding a uniform bit string give rises to a distribution for M similar to the target distribution. For prime numbers, Juels and Ristenpart give a secure DTE that essentially converts a sampling algorithm for uniformly distributed prime numbers into a DTE encoding and decoding algorithm pair.

Given DTEs suitable for password vaults, an HE-based approach has several benefits over the hide-in-an-explicit-list approach of Kamouflage. First, regardless of the quality of the DTE, because there is only one ciphertext, the amount of offline cracking effort required by an attacker is never less than for a conventional PBE-only ciphertext. This means that, unlike Kamouflage, an HE-based scheme will never provide attackers with a speed-up in offline work. Additionally, the size of ciphertexts in HE does not depend on the number of decoys possible, rather decoys are generated "on-the-fly" during a brute-force attack for each guessed master password. Therefore, given a good DTE, the online work required by an adversary is essentially the strength (guessing entropy) of mpw. A strong mpw will mean the attacker must make many online queries.

We would like a cracking-resistant vault to support use of both computer-generated and human-chosen passwords. The former is relatively straightforward, as computer-generated passwords come from an easy-to-characterize distribution. Human-chosen passwords present a significant challenge, however, as they raise the question of whether one can build DTEs that accurately model natural language-type distributions. We show in the next section that such modeling is possible, and handle further challenges of building a full-fledged, encrypted vault management service in the sections that follow.

2.5 Natural Language Encoders for Passwords

Formally, a DTE is a pair of algorithms $\mathsf{DTE} = (\mathsf{encode}, \mathsf{decode})$, where encode is randomized and decode is deterministic. In our context, encode takes as input a vector of passwords $\vec{w}$ and outputs a bit string of some length s. The deterministic decoder decode takes as input an s-bit string and outputs a password vector. We require that the DTE be *correct*, meaning that $\mathsf{decode}(\mathsf{encode}(\vec{w})) = \vec{w}$ with probability 1 over the coins used by encode. Our DTEs will be designed so that the length l of outputs of encode depends only on ℓ, the number of passwords in $\vec{w}$.

As an example, it is simple to construct a DTE for uniformly random, fixed-length strings of symbols drawn from an alphabet Σ that consists of the 96 ASCII printable characters. Encoding works in a character-by-character manner on an input string $x_1 x_2 \ldots x_k$, where $x_i \in \Sigma$. Let $\overline{x_i}$ denote the position of x_i in Σ under some canonical ordering of Σ. Then for each symbol x_i in turn, encode outputs a large (e.g., 128-bit) integer X_i selected randomly subject to the constraint $X_i \bmod 96 = \overline{x_i}$. (See Appendix A.2 for details on security bounds and other considerations.) Decoding operates in the natural way: Given input $X_1 \| X_2 \| \ldots \| X_k$, it yields output $x_1 \| x_2 \| \ldots \| x_k$ such that $\overline{x_i} = X_i \bmod 96$. Straightforward extensions that we omit for brevity allow construction of a DTE over passwords that conform to standard password-composition policies (such as needing at least one integer, one special symbol, etc.). We refer to this DTE as UNIF.

We will use such a simple *uniform DTE* for computer-generated passwords later. But it provides poor security for human-selected passwords, which are clearly not distributed uniformly. This observation brings us to one of our core tasks: building DTEs that securely encode samples from distributions of natural

language-type text. Because we feel that such DTEs will be of broad use, we give them a special name: *natural language encoders*, or NLEs. We focus on DTEs for messages consisting of a single password and, later, lists of passwords. We note, however, that our constructions are quite general and may be applicable in other natural language contexts.

2.5.1 NLEs from password samplers.

A starting point for our NLE constructions is password crackers. Early crackers, such as John the Ripper, simply have stored dictionary lists of popular passwords, and can produce samples in order of likelihood. More modern crackers instead learn compact representations of password distributions from password leaks [39, 40], and permit efficient sampling of passwords over the distribution model.

In a bit more detail, we can view a sampling model for passwords as a deterministic algorithm Samp that takes as input a uniformly random bit string U of sufficient length (often called the "coins") and produces a password w. We can characterize Samp in terms of a distribution p, meaning that a password w is output by Samp with probability $p(w)$ (over the selection of bit string U). The goal of a password cracker is to learn an algorithm Samp from one or more password leaks whose corresponding distribution p closely approximates that of human-generated passwords seen in practice.

We might hope, a priori, that one can build a secure DTE from any sampling algorithm Samp. Unfortunately this seems unlikely to work, in the sense that there exists (admittedly artificial) Samp for which building a DTE appears intractable. Briefly, let $\mathsf{Samp}(U) = \mathsf{H}(U)$ for some cryptographic hash function H, where U

is a random bit string. Then the natural approach for DTE construction is to set $\mathsf{decode}(U) = \mathsf{H}(U)$. But for such decode, correctness would mandate that $\mathsf{encode}(w)$ somehow can sample from the set $\mathsf{H}^{-1}(U)$ of preimages of U, which would contradict the hash function's security. Of course there may be other ways to build $\mathsf{encode}, \mathsf{decode}$ that use Samp only as a black-box, and yet achieve correctness and security. We conjecture that a full counter-example exists, but do not have proof.

Such artificial counter-examples aside, for various classes of Samp we can in fact use the straightforward approach of having $\mathsf{decode}(U) = \mathsf{Samp}(U)$. The only requirement is that we can build $\mathsf{encode}(w)$ that samples uniformly from $\mathsf{Samp}^{-1}(w)$. We show how to do so below for a couple of useful classes of samplers: n-gram models and probabilistic context-free grammar (PCFG) models. We start with the single password case for the different models, and then discuss extensions to the (trickier) case of a vault of possibly related passwords.

2.5.2 NLEs from n-gram models.

So-called n-gram models are used widely in natural language domains [41, 47, 48]. For our purposes, an n-gram is a sequence of characters contained in a longer string. For example, the 4-grams of the word '`password12`' are {`pass, assw, sswo, swor, word, ord1, rd12`}. In building models, it is convenient to add two special characters to every string: `^` to the beginning and `$` to the end. Given this enhancement, the 4-gram set in the example above would also include `^pas` and `d12$`.

An n-gram model is a Markov chain of order $n-1$. The probability of a string

is estimated by

$$\Pr[\, x_1 x_2 \cdots x_k \,] \approx \prod_{i=1}^{k} \Pr[\, x_i | x_{i-(n-1)} \cdots x_{i-1} \,]$$

where the individual probabilities in the product are calculated for a given model empirically via some text corpus. For example, for the RockYou password leak, we let $c(\cdot)$ denote the number of occurrences of a substring in the leak. The empirical probability is then

$$\Pr[\, x_i | x_1 x_2 \cdots x_{i-1} \,] = \frac{c(x_1 x_2 \cdots x_i)}{\sum_x c(x_1 \cdots x_{i-1} x)}$$

for any string $x_1 \cdots x_i$ of any length i. Let $F_{x_{i-(n-1)} \cdots x_{i-1}}$ denote the CDF associated the probability distribution for each history. Then the Markov chain associated to such an n-gram model is a directed graph with nodes labeled by n-grams. An edge from node $x_{i-(n-1)} \cdots x_{i-1}$ to $x_{i-(n-2)} \cdots x_i$ is labeled with x_i and $F_{x_{i-(n-1)} \cdots x_{i-1}}(x_i)$. To sample from the model one starts at node $\hat{}$, samples from $[0, 1)$, finds the first edge[5] whose CDF value is larger than the sample, follows it to move to the next node, and repeats. The process finishes at a node having the stop symbol. The sequence of x_i values seen on the edges is the resulting string.

Note that such a Markov chain may not have edges and nodes sufficient to cover all possible strings. For use in encoding, then, we extend the Markov chain to ensure that each node has an edge labeled with each character. We set the probabilities for these edges to be negligibly small, and re-normalize the other edge weights appropriately. If this implies a new node we add it as well, and have its output edges all have equal probability.

We can build a DTE by encoding strings as their path in the Markov chain. To encode a string $p = x_1 \cdots x_k$, process each x_i in turn by choosing randomly from the values in $[0, 1)$ that would end up picking the edge labeled with x_{i+1}.

[5] We again assume an ordering on edges for which CDF values are strictly increasing.

Decoding simply uses the input as the random choices in a walk. Both encoding and decoding are fast, namely $\mathcal{O}(n)$ for a password of length n.

In our experiments, we use a 4-gram model trained from RY-tr. We denote the resulting DTE by NG. We also explored 5-gram models, but in our experiments these used up more space without a significant improvement in security.

2.5.3 NLEs from PCFG models.

A PCFG is a five-tuple $G = (N, \Sigma, R, S, p)$ where N is a set of *non-terminals*, Σ is a set of *terminals*, R is set of *relations* $N \to \{N \cup \Sigma\}$, $S \in N$ is the *start symbol*, and p is a function of the form $R \to [0, 1]$ denoting the probability of applying a given rule. We require that for any non-terminal $X \in N$, it holds that $\sum_{\beta \in N \cup \Sigma} p(X \to \beta) = 1$. PCFGs are a compact way of representing a distribution of strings in a language. Each derivation for a member of the language defined by the underlying CFG has a probability associated to it.

Weir, Aggarwal, de Medeiros, and Glodek [39] were the first to apply PCFGs to the task of modeling password distributions. They constructed a password cracker that could enumerate passwords (in approximate order of descending probability) in a way that ensured faster cracking compared to previous approaches like John the Ripper. Weir et al. parsed passwords into sets of contiguous sequences of letters, digits or symbols. Further improvements are possible by employing their approach with alternative parsing schemes. Jakobsson and Dhiman [46] and later Veras et al. [40] used a (so-called) maximum coverage approach for parsing passwords with the help of external language specific dictionaries. Veras et al. also used the semantic meaning of passwords to provide finer granularity parsing, and improved

PCFG cracking performance.

We now show how to build a DTE for a single password from any PCFG model. Intuitively, the encoding of a password will be a sequence of probabilities defining a parse tree that is uniformly selected from all such giving rise to the same password. Decoding will just emit the string indicated by the encoded parse tree. We first fix some definitions. A rule $l \to r$ can be specified as a pair (l, r), where l is a non-terminal and r is a terminal or non-terminal. Every edge in a parse tree is a *rule*. A *rule set* is a lexicographically ordered set of rules with the same left-hand-side. A *rule list* is an ordered list of rules generated by depth-first search of the parse-tree of a string / password (with siblings taken in left to right order).

A CFG is completely specified as a set of rule sets. A PCFG is completely specified by what we call here an *admissible* assignment of probabilities to CFG rules. Let $\mathcal{S}$ be a rule set of size $|\mathcal{S}|$ and $p_{\mathcal{S}}(l \to r)$ be the probability assigned to a rule $l \to r$ in $\mathcal{S}$. An admissible assignment of probabilities has the property that $\sum_{(l \to r) \in \mathcal{S}} p_{\mathcal{S}}(l \to r) = 1$. We refer to the probability distribution over rules in a rule set $\mathcal{S}$ for a given admissible assignment as its *induced* probability distribution.

As a technical modification to such a PCFG, we add a special *catch-all rule*. Its left-hand side is the start symbol and its right hand side represents any string. We assign this catch-all rule a very low probability (and normalize other probabilities accordingly). This rule ensures all passwords can be parsed (and generated) by the PCFG model and that the model will never fail to encode any real password.

For a given PCFG, a parse tree, and thus a string str, may be specified as a sequence of probabilities $p_1, \ldots, p_k$ (for sufficiently large k). To construct this parse tree, a rule is selected using p_1 from the rule set for the start symbol S,

producing the children of S in the tree. A rule from the rule set for each child is then selected using p_2, p_3, etc., from left to right. Recursing in this way produces the full parse tree; its leaves, read left to right, constitute str.

As shown in Appendix A.2 we can represent a rule-set probability by an b-bit integer. It follows that a parse tree for a PCFG, and thus a generated string P, may be completely specified by a vector $\vec{X} = \langle X_1, \ldots, X_k \rangle$ of k integers, where $X_i \in \{0,1\}^b$. This vector is not necessarily unique: There may, of course, be multiple vectors corresponding to str.

We can now build a DTE from any PCFG model. Decoding takes as input a vector $\vec{X}$ of integers, uses it to determine a parse tree, and outputs the corresponding password P. This requires time $\mathcal{O}(n \log s)$ for n-character passwords and where s is the size of the largest rule set in the PCFG. Encoding takes as input a password P and selects uniformly at random a vector $\vec{X}$ from the set of all that decode to P. This inverse sampling can be efficiently implemented by finding all parse trees (also known as parse forest) of P, and picking one at random. This is an $\mathcal{O}\left(k^3\right)$ time operation [49, 50]. Note that $\vec{X}$ is of a fixed size k; thus encoding pads out the resulting vector with random bit strings representing sufficiently many extra integers.

Of course, all of the above relies on having a PCFG that accurately models the password distribution, a research topic in its own right [39, 24, 40, 41]. Our general approach has the benefit of allowing us to use any of these prior PCFG construction approaches. We built our own hand-tuned PCFG using the RockYou training set, employing a combination of techniques from Weir et al. [39] and Veras et al. [40]. In initial evaluations it performs better than the Weir et al. PCFG (in terms of security; see Section 2.6). Some further details on the process for gener-

ating it are provided in Appendix A.1. We refer to the DTE built from our new PCFG as PCFG. As baselines for decoy generation quality, we built two additional DTEs, WEIR and WEIR-UNIF. WEIR uses the grammar proposed by Weir et al. [39]. WEIR-UNIF is the same grammar except it ignores frequency information and treats all rules inside a rule-set with equal probability. These grammars are functionally equivalent to those used by Kamouflage+ and Kamouflage, respectively, when restricted to a single password.

2.5.4 From one-password DTEs to vault DTEs.

We can easily extend any single-password DTE to handle multiple passwords by applying an encoder independently to each password in the vault. This models a vault distribution in which passwords are independent of one another. This is especially useful when we have both computer-generated passwords in a vault as well as human-chosen; we can choose appropriate single-password DTEs for each case. We denote this independent-password DTE by MPW-DTE (for multiple passwords).

Such a DTE may not work well when users repeat or have related passwords in their vaults, however, motivating a decode algorithm that generates a vector of passwords $\vec{w}$ in which passwords repeat in full or part. We introduce a technique for embellishing PCFG-based single-password DTEs to handle vaults in this way, what we refer to as the *sub-grammar approach*. The intuition is that if DTE-decode samples passwords from a smaller domain than the actual trained PCFG, it will often end up using the same password components or full passwords.

In more detail, SG-DTE (for sub-grammar) is the following DTE scheme. En-

coding first parses all the passwords in $\vec{w}$ using the trained PCFG. It then generates a new sub-grammar PCFG that consists of the cumulative set of rules used in parsing the passwords in $\vec{w}$. The rule probabilities are copied from the original PCFG and then normalized over the sub-grammar PCFG. (We also copy special rule sets described in detail in Appendix A.1. For example, `T`, the catch-all rule, is always included in the sub-grammar.) This sub-grammar is encoded as the first part of the DTE output, as detailed below. Finally, the DTE separately encodes each $w \in \vec{w}$ as in PCFG, but using the sub-grammar PCFG.

Decoding works in the natural way: first decode the sub-grammar PCFG, then decode the encoding of each password using the resulting sub-grammar PCFG.

Encoding/Decoding of the sub-grammar. Given a canonical representation of the trained PCFG, a sub-grammar can be specified by simply encoding for each non-terminal (except `T`) the number of corresponding rules used by the sub-grammar followed by a list of such rules. Each rule in the list is encoded in the same way as a derivation rule for a password.

To encode the size of a rule set we proceed as follows. Using a set of leaked password vaults in Pastebin (see Section 2.6.1), we generate the sub-grammar PCFG for all the vaults of size ≥ 2. For each non-terminal in the PCFG (except `T`), we then create a histogram of the number of the non-terminal rules used by each sub-grammar. This gives a per-non-terminal empirical distribution on the number of rules used, which we use as the distribution for sizes that should arise when decoding a random string to a sub-grammar. The DTE encodes this distribution via the inverse transform sampling mechanism of [25].

We have explained now how SG-DTE encodes an input $\vec{w}$ in a way that captures

structure across passwords, making SG-DTE suitable for encoding of password vaults. One additional step is required in the full specification of encode: SG-DTE pads out all encodings to a constant length with random bits. This is important because the size of the encoding will otherwise leak the size of the sub-grammar.

2.6 Evaluating the Encoders

We have shown how to construct functional NLEs that model real-world password selections by human users of password vaults. To evaluate the quality of these NLEs, we now study their resilience to attack using standard machine-learning techniques.

Recall that in an offline brute-force attack the adversary makes repeated guesses at the master password and decrypts the target vault under each guess. The task of the adversary is to identify the result of decryption under the true master password, i.e., to determine the true plaintext for the vault.

Suppose q is the number of such guessing / decryption attempts. If the true master password is among the adversary's guesses, the result will be $q - 1$ random samples from the NLE, as well as the true plaintext, and thus q plaintext candidates in total.

We consider an adversary that orders these q plaintexts in a list from highest to lowest likelihood of being the true vault (in the adversary's view). The adversary's best strategy for attacking the vault is then to make online authentication attempts using one password from each plaintext (decrypted vault) in order from the list. Thus the position of the true plaintext vault in the list indicates the number of

online authentication attempts the adversary must make. We evaluate such an attack for an adversary that ignores master password likelihood, and instead uses machine learning (ML) algorithms on the plaintexts to order the list.

Evaluating single decoy passwords. We start by evaluating the security of NLEs for single decoy passwords, and leave full vault analysis to the next subsection. The security goal for a single-password NLE is to produce a decoy password that is indistinguishable by an adversary from a true, user-selected one. We evaluate security in two ways.

First, we look at the accuracy with an binary adversarial classifier can assign the right label ("true" / "decoy") to a password. Second, we evaluate the ability of such a classifier to assign a high rank to a true password in an ordered list of plaintexts (single passwords) as described above. For this second evaluation, we use the confidence measure of the classifier for a label assignment of "true" as the basis for ranking passwords in the list.

Methodology. We explored a number of approaches to attack, and settled on building machine learning (ML) classifiers to distinguish between true and decoy passwords. We treat this as a supervised learning problem. That means we train a classifier with two sets: labeled true passwords and labeled decoy passwords. We test by drawing from two (disjoint) sources of real passwords and decoy passwords. After experiments with a number of feature and classifier types, we have chosen to report only on the best-performing option, random forest classifiers [51] with 20 estimators using the following features:

(1) *Bag of characters:* This feature captures the frequency distribution of characters in a password. We represent this feature as a vector of integers of size

equal to the number of printable characters. We also append the length of the password to the vector.

(2) *n-gram probability:* We train two 4-gram models separately on each of the two classes of password (true and decoy) provided for training. For a given password, we use the probability yielded by each of these two models as a feature. (These two probabilities / features do not sum in practice to 100%, as they would for perfectly complementary models.)

We apply this classifier to the various training set / testing set pairs explored in our experiments.

We evaluate five distinct NLEs as sources of decoys: WEIR-UNIF, WEIR, UNIF, NG, and PCFG. These are all trained using the RockYou training set RY-tr. To generate a decoy using any of these NLEs, we decode a fresh, random bit string of suitable length. As a sixth source of "decoys" we sample directly from RY-tr.

As sources of true passwords, we use the RY-ts, Myspace, and Yahoo data sets. Use of RY-ts creates a case where the NLE is trained using samples from the same data set (but not the same data) as the classifier is tested upon. Use of Myspace and Yahoo data sets creates a case where true passwords originate from a different distribution, which we expect to make the task of distinguishing true from decoy easier for the adversary.

As we have six sources of decoy passwords and three sets of true passwords, we have a total of eighteen true / decoy source pairs on which to conduct our experiments.

For each experiment, given a true / decoy password source pair, we first sample t passwords from the true data set uniformly without replacement to obtain a

NLEs	Myspace		Yahoo		RY-ts	
	α	$\bar{r}$	α	$\bar{r}$	α	$\bar{r}$
WEIR-UNIF	66	24	72	13	82	5
WEIR	63	35	54	36	60	25
UNIF	86	2	97	<2	97	<2
NG	70	22	68	41	61	41
PCFG	70	26	58	39	57	39
RY-tr	70	22	64	50	50	50

Figure 2.4: For different decoy / true password source pairs, percentage classification accuracy (α) and percentage average rank ($\bar{r}$) of a real password in a list of q = 1,000 decoy passwords for ML adversary. Lower α and higher $\bar{r}$ signify good decoys.

derived set of true passwords. We set $t = 100,000$ or the size of the true data set, whichever is smaller. We treat the sampled data set as a multiset, meaning that the probability of selecting a password is proportional to the number of times it appears in the set. We also treat the derived data set as multiset, meaning that a given true password can be sampled and thus appear multiple times. We then generate a set of t decoy passwords using the decoy source, i.e., by using the appropriate NLE or sampling from the "decoy" set RY-tr. Using the resulting pair of derived data sets, we do a 10-fold cross-validation of the random forest classifier with 90% / 10% training / testing splits.

For our experiments in true / decoy password classification, we measure α, the average accuracy of the classifier on testing data. In those experiments involving ranking of q passwords in order of likelihood of being the true password, we order passwords according to the confidence of the classifier in assigning a "true" label. We measure $\bar{r}$, the average rank of the true password in the resulting list. (Thus $\bar{r}$ is an estimate of the number of online authentication attempts required for a brute-force attacker that uses the classifier to identify the true password.)

An effective classifier will achieve a high value of α and a low value of $\bar{r}$. For example, a classifier that performs no better than random on a given decoy

generation algorithm will on expectation achieve $\alpha = 50\%$ and $\bar{r} = (q+1)/2$. A perfect classifier will achieve $\alpha = 100\%$ and $\bar{r} = 1$.

Figure 2.4 reports the average classification accuracy (α) and average rank ($\bar{r}$) (expressed as a percentage) across our eighteen true / decoy password source pairs. For experiments, we set $q = 1{,}000$. In other words, we drew one password from the true password set and inserted it in a randomly selected position among 999 decoys generated from the decoy source. (We chose $q = 1{,}000$ as larger values, e.g., $q = 10{,}000$, yielded similar results in preliminary experiments, but resulted in significantly longer times generating decoys and thus for overall experiment execution.)

Several outcomes of our experiments are notable. As expected, the uniform NLE UNIF does quite poorly, with the classifier strongly distinguishing it from true passwords. Also as expected, the adversary performs better in nearly all cases against Myspace and Yahoo data than RY-tr, that is, when the adversary trains on the true password source, and the decoy generator designer does not. (As a sanity check, given the use of RY-tr as a source of "decoy" passwords, and RY-ts as a source of true passwords, i.e., a common source for both, the adversary does no better than random guessing in distinguishing true from "decoy.")

It is important to observe that no decoy generator is consistently superior to others across the board. For example, WEIR resists attack best for Myspace and Yahoo data sets, while PCFG is superior in the case of RY-ts. As all decoy generators are trained on RY-tr, these results suggest that WEIR generalizes better than PCFG, in the sense that it can be deployed effectively to protect password sources different from those on which it has been trained. Strikingly, in the task of distinguishing decoys from true passwords drawn from the the Myspace and Yahoo

data sets, WEIR generates decoys that are harder for the adversary than "decoys" (true passwords) from RY-tr.

2.6.1 Evaluating complete password vaults

We now describe our evaluation of SG-DTE, our NLE for generating full decoy vaults. Our evaluation relies on a set of leaked password vault contents that we obtained from an anonymous public post to Pastebin. We refer to this data set as Pastebin. Pastebin appears to have been gathered via malware running on a number of clients, and is thus suggestive of the kind of data an adversary might exploit. A limitation of the data set is that it takes the form of unordered username / password pairs. Thus the only way to ascertain that two passwords came from the same vault is on the basis of a common or similar associated usernames. We organized the passwords into hypothesized vaults through a manual sanitization procedure on Pastebin whose details we omit for brevity, but which we will publish with our code.

Some statistics on Pastebin are given in Figure 2.5. Here m denotes the number of passwords in a vault.

Adversarial classifier. To construct a classifier capable of distinguishing true vaults from those generated by SG-DTE, we make use of the following four feature vectors:

(1) *Repeat count:* A vector of three vault features: (1) Number of unique passwords, (2) Number of passwords unique up to leet transformation and capitalization, and (3) Number of unique tokens inside a vault. These counts are

m	Statistic	Value
$2 \leq m \leq 3$	Users	100
	Domains	140
	Total no. of unique PWs	155
	Avg. domains/users	2.06
	Avg. PWs/user	1.60
	Median PWs/user	2
$4 \leq m \leq 8$	Users	89
	Domains	329
	Total no. of unique PWs	378
	Avg. domains/users	5.27
	Avg. PWs/user	2.58
	Median PWs/user	2
$9 \leq m \leq 50$	Users	87
	Domains	887
	Total no. of unique PWs	741
	Avg. domains/users	15.01
	Avg. PWs/user	4.21
	Median PWs/user	4
Min. entropy of full data-set		6.13

Figure 2.5: Summary statistics for Pastebin password vault leak. Here, m is the number of passwords in a vault.

normalized by (divided by) the number of passwords in the vault.

(2) *Edit distance:* A vector $\vec{x} =< \vec{x}_1, \vec{x}_2 \ldots \vec{x}_k >$ of size k, where $\vec{x}_i$ denotes the number of passwords pairs within edit-distance i (case insensitive). This feature aims to capture user "tweaking" of passwords across sites, i.e., small modifications made to a password to create unique per-site variants.

(3) *n-gram structure:* A vector $\vec{x}$ of size k, where $\vec{x}_i$ is the percentage frequency of the i^{th} most popular n-gram in the vault, for some n. (We use $n = 5$.) This feature characterizes token reuse among passwords.

(4) *Combined:* By this we denote a combination of all three aforementioned feature vectors.

In all individual vectors, we let $k = 5$. We experimented with several ML engines, including random forest and clustering, but the best-performing was a Support Vector Machine (SVM). We construct four support vector machine (SVM) classifiers with a radial basis function kernel, one for each of the feature vectors

given above.

Our training set consists of an equal number of labeled decoy and true password vaults of equal size (m). We use the Pastebin data set for true vaults. (See the "Users" statistic in Figure 2.5 for the number of available instances.) By analogy with our evaluation of single passwords, the adversarial algorithm runs the SVM on q vaults in turn, and orders these vaults according to the SVM-estimated probability of being in the "true" class.

For each size m, let c_m be the number of true vaults in the Pastebin data-set with m passwords. We perform $(c_m - 1)-$fold cross-validation, meaning we set aside one vault of size m for testing as the true vault and train on all others; we repeat this process for every vault. To generate decoy vaults for training and testing we generate uniformly random bit strings and decode them using the appropriate NLE.

As previously described, the success metric is $\bar{r}$, the average value over all true passwords vaults of the rank r in the adversarially ordered list, expressed as a percentage.

Figure 2.6 summarizes results for two full-vault NLEs. Recall that the one labeled SG-DTE is the sub-grammar NLE built from our PCFG. For comparison, we also consider a naïve one, labeled MPW-DTE, that outputs m independent applications of the PCFG NLE. We set $q = 1,000$. The table on the left compares the different attacks across all vaults of size 2–50 against the two NLEs. As expected, treating passwords as independent in MPW-DTE yields decoys that an attacker can easily distinguish from a true vault, with $\bar{r} = 0.6\%$ for two of our adversarial algorithms. In contrast, our sub-grammar approach SG-DTE achieves $\bar{r} > 30\%$ in

$\bar{r}$ (average rank %)		
Feature used	MPW	SG
Repeat count	0.6	37.4
Edit dist.	1.2	41.4
n-gram	0.6	38.5
Combined	1.0	39.7

$\bar{r}$ (average rank %)		
m	MPW	SG
2–3	13.2	32.8
4–8	0.8	38.0
9–50	0.3	38.9

Figure 2.6: Performance of ML attacks against NLEs. **(Left)** Average rank of the real vault ($\bar{r}$) over all vault sizes ($m \in [2, 50]$) for MPW-DTE and SG-DTE using different feature types. **(Right)** Average rank $\bar{r}$ of the true vault obtained with Repeat-count feature vector, broken down by vault size m.

all of our presented experiments.

The right table compares the performance of the Repeat-count feature vector against MPW-DTE and SG-DTE by vault size. This attack is quite effective against MPW-DTE, confirming the observation that treating passwords within a vault independently poorly models real vaults. This series of experiments also brings to light the fact that the security of the NLE degrades as the vault size m increases. Again, SG-DTE fares much better.

2.7 Honey Encryption for Vaults

Given the NLEs from Section 2.5, we can now build an encryption service for vaults. Our construction uses a similar design as Juels and Ristenpart's [25] honey encryption (HE) construction, which composed a DTE with a conventional password-based encryption scheme. But ours will necessarily be more complicated. We will combine multiple different DTEs (an NLE for human-generated and a regular DTE for computer-generated passwords) and handle side information such as domains in a privacy-preserving manner. We will also need to handle adding new entries to an existing vault and removing entries.

Abstractly, vault encryption takes as input a set of domains $\vec{D} = (D_1, \ldots, D_\ell)$, associated passwords $\vec{w} = (w_1, \ldots, w_\ell)$, and a vector of bits $\vec{h} = (h_1, \ldots, h_\ell)$ for which a 1 signifies that the password was input by a user and a 0 signifies that the password was randomly generated. The client selects randomly-generated passwords by selecting uniformly from a large set (of size about 2^{90}) that includes passwords accepted by all the website policies that we tested.

We now present a basic HE scheme. This scheme hides passwords, but takes the simple approach of storing domains in the clear with the ciphertext.

Basic HE scheme. Upon input $\vec{D}, \vec{w}, \vec{h}$, this scheme proceeds as follows. We first apply the sub-grammar NLE SG to the subset of passwords in $\vec{w}$ for which $h_i = 1$. To each of the remaining passwords with $h_i = 0$ we apply the DTE UNIF. The result from both steps is a bit string $S = S_0 \| S_1 \| \cdots \| S_\ell$ with S_0 the output from encoding the sub-grammar and each S_i either an encoding under PCFG using the sub-grammar ($h_i = 1$) or an encoding under UNIF ($h_i = 0$).

The string S is then encrypted as follows. First, derive a key $K = \mathsf{KDF}(mpw, \mathsf{sa})$ for a freshly generated uniform salt sa and where mpw is the user's master password. Here KDF is a password based key derivation function (PBKDF) that is strengthened to be as slow as tolerable during normal usage [35]. Then, encrypt each S_i (for $i = 0$ to ℓ) independently using AES in counter mode with key K and a fresh random IV. This produces a sequence of $\ell + 1$ CTR-mode ciphertexts $\vec{C} = (C_0, \ldots, C_\ell)$. The final vault ciphertext includes a (conventional) encoding of $\vec{D}, \vec{h}, \mathsf{sa}, \vec{C}$. Decryption works in a straightforward way.

This HE scheme is relatively simple (once the NLE and DTE are fixed) and space efficient. However, $\vec{D}$ and $\vec{h}$ are stored in the clear and this means that

attackers obtaining access to the ciphertext learn the domains for which the user has an account as well as which passwords were randomly generated. One approach to rectify this would be to specify a DTE for encoding $\vec{D}$. One could use popularity statistics for domains, for example based on Alexa rankings. However note that this may sacrifice security against offline brute-force attacks in the case that an attacker knows with high certainty the set of domains associated with a user's vault.

This highlights a delicate challenge in the use of HE: if an attacker can easily obtain knowledge about a portion of the plaintext, it may be better to not apply HE to that portion of the plaintext. We may view domain information as such easily-obtainable side information that it is not worth encrypting. To provide domain privacy, though, we must do more. We now describe two approaches: HE-DH1 and HE-DH2.

HE-DH1. In this scheme, we hide an individual user's domains in the set of all domains used by users and include in each user's vault "dummy" entries for unused domains. To achieve privacy of domains the goal will be that an attacker cannot distinguish between a dummy and real entry. In the following, we fix a set of popular domains $\vec{D}^* = (D^*_1, \ldots, D^*_{s_1})$. We require that $\vec{D}^*$ be a superset of all the domains used by users. We discuss how to achieve this momentarily.

To encrypt an input $\vec{D}, \vec{h}, \vec{w}$ we first encode the passwords as described above for the basic scheme: apply SG to the set of human-generated passwords and UNIF to each of the computer-generated passwords. For any domain in $\vec{D}^*$ but not in $\vec{D}$, we generate a dummy encoding as follows. First we choose a bit h to select whether this domain should have a human-generated dummy entry or a computer-generated one. We discuss how to bias this bit selection below. Then we

generate a random bit string of length equal to the length of outputs of PCFG (if $h = 1$) or UNIF (if $h = 0$). We then generate the full encoding $S = S_0 \| S_1 \| \cdots \| S_{s_1}$ by inserting the per-password encodings or dummy encodings to match the order from $\vec{D}^*$. We then encrypt S as in the basic scheme.

The distribution alluded to above for dummy encodings does not affect confidentiality of mpw or $\vec{w}$, but rather confidentiality of the domains associated to a user and whether the user has human or generated passwords for each such domain. One can, for example, set this distribution to be biased towards human-generated passwords.

Note that decryption with either the correct mpw or an incorrect one produces s_1 passwords. The $s_1 - \ell$ honey passwords corresponding to the dummy entries obscure from an attacker which sites are in use by the user (even in the extreme case that the attacker has guessed mpw somehow). When mounting brute-force attacks, the dummy entries might hinder attempts to perform online checks of recovered passwords. In particular, if a domain's login web page follows best practices and does not leak whether a user has an account there (regardless of correctness of the provided password), then the attacker may not be able to distinguish between the situation in which a certain domain is not used by a user and the situation in which the decryption attempt resulted in a honey password.

The downside of the above approach is that s_1 may need to be very large and for each user the storage service (described below in Section 2.8) must store $\mathcal{O}(() s_1)$ bits of data. We can grow s_1 over time by having clients inform the service of when a new domain should be added to D^* and the server can insert dummy entries in previous vaults by just inserting random bit strings in the appropriate location for each vault ciphertext. (this is possible because a separate CTR-mode encryption

is used for each vault entry.) When the system is first setup, an initial relatively small popular domains list can be seeded with highly popular domains.

Another approach to reducing overheads is to bucket users into separate groups, each group having their own popular domains list. This enables tuning the size of vaults relative to per-group domain confidentiality.

HE-DH2. In this scheme, we adopt an alternative approach to dealing with the long tail of domains, and use a honey-encrypted overflow table. Fix some number $s_2 > 0$. For each of the domains not in the current popular domain set, we use the following procedure. First apply the PCFG (using the sub-grammar or UNIF appropriately to the password to get a bit string S'. Then hash the domain name and take the result modulo s_2 to yield an index $j \in [0..s_2 - 1]$. Set $S^*_{s_1+j+1}$ to S'. Some indices in $[0..s_2 - 1]$ will be unused after handling all domains outside the popular domain set; we fill these with dummy encodings. The additional s_2 seeds are each encrypted with CTR mode, making the final, full ciphertext $\vec{D}^*, \vec{h}^*, \mathsf{sa}, \vec{C}$. Note that now, $\vec{C}$ contains $s_1 + s_2 + 1$ individual CTR-mode encryption ciphertexts for the sub-grammar and $s_1 + s_2$ individual, possibly dummy, passwords.

By setting s_2 large enough relative to the expected number of domains not in D^* we can ensure that with high probability no two domains hash to the same location. Note that the domains associated with the overflow table are not stored with the ciphertext. To decrypt, the requested domain is checked to see if it is in D^* and if not it is hashed to find the appropriate entry in the last s_2 HE ciphertexts.

Updating a vault. To update a password for a particular domain in the basic scheme or HE-DH1, one first decrypts the entire vault, changes the appropriate entry, and then encrypts the modified vault with fresh randomness (including the

salt). needed to ensure the sub-grammar is consistent with the encoded content. For HE-DH2, one proceeds much the same, also decrypting each of the s_2 entries in the overflow table. The appropriate domain's entry is updated (found either by looking in the popular domains list or, failing that, hashing the domain to be updated to find it in the overflow table). Finally, the modified vault is encrypted (with fresh randomness).

Deletion of a password can be performed by converting the appropriate entry into a dummy entry while also updating the sub-grammar by removing any now unnecessary rules.

Security discussion. Our primary goal is confidentiality of the plaintext passwords. All passwords are first encoded using an appropriate DTE and then encrypted using a PBE scheme. Should the user's master password be strong, even an offline brute-force attack is infeasible and, in particular, it will require as much work to break any of the schemes above as would be to break a conventional PBE encryption. Should the user's master password be weak, then by construction decrypting the ciphertext under any incorrect master password gives back a sample from the DTE distribution. In particular, we believe there to be no speed-up attacks that allow the attacker to rule out a particular incorrect master password without having to determine if the recovered plaintext is decoy or not. As we showed in the previous section, our NLE is good enough that distinguishing human-generated passwords is challenging even for sophisticated adversaries.

The above is admittedly informal reasoning, and does not rule out improved attacks. We would prefer a formal analysis of plaintext vault recovery security akin to those given for simpler honey encryption schemes in [25], which would reduce security to solely depend on DTE quality. Those techniques rely on a closed-form

description of the distribution of password vaults as produced by decoding uniform strings. Unfortunately we do not know how to determine one; even estimating the distribution of single passwords is impractical with sampled data [36]. Formal analysis remains an interesting open question.

If an attacker obtains the encryptions of a vault before and after an update, then security falls back to that of conventional PBE. One simply decrypts both vaults under each guessed master password, and with high probability the contents of the two plaintext vaults will match (except where updates occurred) with high probability only with the correct master password. This is a limitation of all decoy-based approaches we are aware of and finding a solution for update security is an interesting open question.

Another security goal is domain hiding. As discussed earlier, adding dummy ciphertexts (random bit strings) for the latter two schemes for unused domains means that an offline attacker will recover passwords for these domains as well. The same reasoning extends to the use of the overflow table. The complexity of the sub-grammar may leak some information about the overall number of human-generated passwords in-use, but not which of the domains marked as having human-generated passwords are dummy encodings.

2.8 The NoCrack System

We now turn to the design of the full honey-vault service that we call NoCrack. Our architecture closely follows deployed commercial systems, such as LastPass[6]. A web-storage service exposes a RESTful web API over HTTPS for backing up user

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

vaults and synchronizing vaults across devices. To achieve the security benefits of HE, however, we must design this service carefully.

The challenge of password-based logins. One encounters an interesting challenge when attempting to build a decoy-based system which supports backup of user vaults: how to authenticate users to the service that is responsible for backing up their vaults. In particular, the status quo in industry is for users to choose a username and service password. The password would be sent over HTTPS to the server, hashed, and stored to authenticate future requests. But customers are likely to choose this service password to be the same as their vault's master password. If an attacker compromises the storage service and obtains both a user's encrypted vault and the service password hash, they can mount a brute-force attack against the service password hash, learn the service password, and then decrypt the vault.

One might attempt to mitigate with this by securing the password hash separately from vaults. Or one could avoid backup of encrypted vaults entirely, but this would leave users responsible and violate our goal of matching features of existing services. We therefore go a different route, and forego password-based login to the storage service completely.

Device enrollment. A new user registers with the service by providing an email address (also used as an identifier), to which a standard proof-of-ownership challenge is sent. To hinder abuse of the registration functionality, the service can rate limit such requests and require solution of an appropriate CAPTCHA [52]. The proof-of-ownership is an email including a randomly generated 128-bit temporary token (encoded in Base64 format, 22 characters long). The user copies this temporary token into the client program which submits the token over an enroll API

call. The server verifies the temporary token, and returns to the client program a (long term) bearer token (also 128 bits) that can be used as a key to authenticate subsequent requests using HMAC. At this stage the client device is enrolled. Note that all communication is performed over TLS.

Additional devices can be enrolled in a similar manner by having an already-enrolled client device to generate a token for the new device or sending a new temporary token via email. Should a user lose all access to a device with a current bearer token, they can easily obtain a new token via the same enrollment process.

We note that two-factor authentication would be straightforward to support by requiring a proof-of-ownership of a phone number or a correct hardware token-generated one-time password to obtain a device bearer token.

Synchronizing with the server. An enrolled client device can compare their local information with that stored under their account on the server. This involves ensuring the client and storage service have the same version of the vault, which, in normal usage, is cached on the client device. To save bandwidth, downloads and uploads can be done in an efficient manner via any standard "diff" mechanism — in particular our HE schemes support sending only portions of the ciphertext at a time.

The client. We currently have only a command line client supported, but future versions could easily integrate with popular browsers via an extension. The client caches the vault locally, but never stores it in the clear on persistent storage. The client queries the service when run to determine if it needs to synchronize the vault. At the beginning of a browsing session, the user is prompted for the master password and the vault is decrypted. To check for typos, we can use dynamic

security skins [53] (as suggested also for use with Kamouflage), which show a color or picture that is computed as a hash of the master password (but never stored). The output of the KDF can be cached in memory in order to decrypt individual domains as needed, while the master password itself is expunged from memory immediately.

Note that the HE scheme does not handle login names; we assume that browser caching mechanisms can handle this for a user if they desire. Should a login detectably fail for the user due to master password typo and the user does not observe the incorrect security skin, the client can prompt the user to reenter their master password. By construction, there might be dummy password entries in NoCrack for some domains where the user does not have an account. The user and/or the browser is responsible to distinguish the domains where the user has an account.

Implementation and performance. We implemented a prototype of NoCrack in Python-2.7. On the server side we used Flask and Sqlite3. To normalize domains we use the Python Public-Suffix library. All cryptographic operations use PyCrypto-2.6.1. We use AES within CTR mode encryption, and SHA-256 within PBKDF2 for key derivation. Many of the operations are parallelizable; we use the Python multiprocessing library for this but note that our prototype implementation does not yet fully take advantage of parallelization. The client and server consist of 3,102 total lines of code as counted by the utility cloc (not counting libraries). All experiments were performed on an Intel Core-i5 with 16 GB of RAM running Linux.

We provide some basic performance numbers for our most complex honey encryption scheme HE-DH2, but emphasize that this is a naive implementa-

Operation	$s = 2$	200	2,000	20,000
Recover password	6.34 ms	6.41 ms	6.42 ms	6.50 ms
Add password	0.13 s	0.68 s	1.11 s	9.25 s
Vault size on disk	4.71 KB	164.00 KB	1.55 MB	15.26 MB

Figure 2.7: Running times (median over 100 trials) of operations for different vault sizes $s = s_1 + s_2$. The final row is size of encrypted vaults on disk.

tion and some improvements will be easy. We fix various vault sizes $s \in \{2, 200, 2{,}000, 20{,}000\}$ and set $s_1 = s_2 = s/2$ (these are the sizes of the popular domains table and overflow table, respectively). We used integer representation size $b = 128$. for encoding fractions. We start by generating a random ciphertext of size appropriate for the values of s_1, s_2 assuming some short arbitrary domain size and that all passwords are human generated (the worst-case for performance). We then measure the time to recover a particular vault password as well as to add a password to the vault. We report in Table 2.7 the median times over 100 trials. Variance in timing was negligible.

Time for recovering a single password is fast, and agnostic to the size of the vault. This is because our design allows random access into the vault. Time for adding passwords increases with s, since our scheme decrypts and decodes all s entries, updates the new password, then re-encodes and re-encrypts all s entries (this is required to keep the sub-grammar synchronized with the contents of the vault.) The bulk of the time is spent in encoding and re-encoding passwords. This operation is still only around one second for large vaults, and large vaults are needed only to support domain hiding. The encrypted vaults are also of reasonable size. We conclude that, while NoCrack does incur time and space overheads relative to conventionally encrypted vaults, the absolute performance is more than sufficient for the envisioned usage scenarios.

2.9 Related work

Honey objects. The use of decoy objects such as honeypots or decoy documents [54, 55] is well-established in information security practice. More closely related to our work here are honeywords [56], decoy passwords associated with each user in a password database. The honeywords system involves fake individual passwords, rather than password sets, and does not help with decoy security for password vaults, our goal here.

We also note that decoy document and honeyword systems are distributed: they assume explicit storage of secrets that distinguish decoy from real objects in a trustworthy location (a "honeychecker") separate from the system containing the decoy objects. See [57] for a discussion of the distinction between such systems and those in which these secrets (e.g., master passwords) are provided by a user, as in NoCrack.

An early decoy system involving encryption under user-furnished secrets was proposed by Hoover and Kausik [38]; it only supports encryption of specially crafted RSA private keys. Honey encryption [25] introduced a general framework for incorporating honey objects into encryption. As explained earlier, it does not prescribe constructions for specific message types, which gives rise to one of the major technical challenges we faced in building NoCrack.

A detailed discussion of Kamouflage [37] was given in Section 4.2.

Password-based key derivation. Key stretching, where one slows down key derivation, was first defined by Kelsey et al. [58], and standardized later in PKCS#5 [35]. Boyen proposed halting password puzzles [59] in which the key-

derivation will run indefinitely on incorrect password guesses and only terminates (after an unspecified length of time) upon correct guesses. Another approach is to incorporate memory-hard functions, which require a significant amount of RAM to compute efficiently, such as done in scrypt [13]. Each of these techniques slows down offline brute-force attacks, but do not force attackers to make online queries.

Stateless password managers. Several schemes exist for strengthening user passwords (and preventing direct password reuse) by hashing a master secret with domain names to dynamically generate per-domain passwords. An early example was the Lucent Personal Web Assistant (LPWA) [60]; later variants include PwdHash [61] and Password Multiplier, a scheme by Halderman et al. [62]. Chiasson et al. conducted a usability study of both PwdHash and Password Multiplier and found the majority of users could not successfully use them as intended to generate strong passwords [63]. Another usability challenge is dealing with sites with a password policy banning the output of the password hash; for this reason NoCrack uses a simple set of rules for computer-generated passwords.

Password managers. In addition to Kamouflage [37], several academic proposals have sought to improve the usability and security of stateful password managers. Passpet [64] generates random passwords per-domain and allows users to assign avatars to different websites to easily identify which passwords are used with which website. Tapas [65] is a prototype two-factor password manager which distributes passwords into shares between a computer and a mobile phone.

Karole et al. [66] performed a usability evaluation comparing three common approaches to password vaults: online services, phone applications, and USB tokens. Interestingly, they found that the online service was by far the easiest for partic-

ipants to use, although participants stated a clear preference for the phone-based solution because most didn't want to entrust all of their passwords to a cloud-based service. These findings are a compelling justification for NoCrack, which enables the convenience of cloud-based password vault backup with higher security against compromise.

CHAPTER 3

PASSWORD TYPOS AND HOW TO CORRECT THEM SECURELY

This chapter was published in IEEE Symposium on Security and Privacy (S&P), 2016 [30].

3.1 Introduction

Despite repeated calls for their demise (cf. [7]), human-chosen passwords remain the primary form of user authentication on the Internet. A long line of investigation has shown that passwords are easily predicted by attackers (cf. [67, 36, 68]), that strength meters offer limited improvements to security [69], that password expiration does not increase security [70], and that users have a hard time remembering complex passwords [71, 72, 73, 69].

A handful of works have pointed out that complex, user-chosen passwords are not only more difficult to remember, but also more difficult to type [74, 75, 72]. But these studies are quite limited, investigating neither the prevalence nor form of typos across a wide user base. Additional anecdotes arise in industry, where a few web services seem to intentionally allow a small set of typos [76, 77, 78, 79]. Facebook currently accepts a password whether or not the user capitalizes the first letter of their password (assuming it starts with a letter), and whether or not they have the caps lock on. But no information about why they do this has been published, and, more importantly, whether this degrades security is unclear.

We provide the first detailed treatment of password typos. We start by measuring empirically the rates and nature of typos made by users. We perform preliminary experiments with Amazon Mechanical Turk (MTurk) in which we task

human workers with transcribing passwords drawn from the RockYou password leak.[1] This does not perfectly model password entry (among other reasons, because the passwords were not the workers' own), but allows us to collect over 100,000 submissions in short order across thousands of workers. Our experiment provides important, basic insights into common typographical errors. We find that a large number are proximity errors (hitting a key near the intended one). Several other common ones are what we call "easily-correctable" typos: they can all be corrected by simple functions applied to the submitted, incorrect password. Examples of the latter include accidentally hitting the caps lock, implementing incorrect first-letter capitalization, adding a character to the front or end of a password, and missing the shift key when entering a symbol at the end of a password. These easily-correctable typos account for 20% of the typos observed in our MTurk study, an observation that serves as a key basis for our work.

Armed with correction functions for easily-correctable typos, we instrument Dropbox's production, Internet-scale login infrastructure. This permits measurement of typo prevalence at scale without changing the way Dropbox currently performs user authentication and with no increased risk of exposure of passwords (i.e., we never store passwords or any information that could help in guessing them). While we cannot reveal the absolute number of requests seen during measurements for reasons of confidentiality, we note that Dropbox has hundreds of millions of customers and all user accounts were instrumented. We first perform a 24-hour measurement to identify login attempts involving the easily-correctable, common typos surfaced in the MTurk study. We find that over 9% of failed login attempts result from just one of three easily-correctable typos (caps lock, first letter case, and adding a character to the end). We perform a subsequent 24-hour

[1]We submitted our experiment design to our IRB, but received an exemption for lack of collecting any PII.

experiment to analyze the impact of correcting just these top three typos. This experiment reveals that *3% of all users failed to login, but could have done so given correction of one of these three easily-correctable typos.* Many other users could have avoided multiple login attempts, significantly decreasing the time required to login. In summary, our measurements suggest that easily-correctable password typos represent a significant burden on users and businesses.

All of this suggests typo-tolerance could bring substantial usability benefits. What remains is to determine if such typo-tolerance necessarily degrades security. The intuition would be that it does, because guesses might cover multiple possible passwords. We show, however, that this intuition is flawed.

We provide a formal framework that enables principled investigation. We define *typo-tolerant password checkers*, and among these a special class that we call *relaxed checkers*. Relaxed checkers are systems that start with an existing exact system (e.g., comparing salted bcrypt hashes or using an encrypted password onion [80]). The system is "relaxed" through a modification that additionally searches a small space of corrections to the submitted password. This search allows easy deployment of typo-tolerance, while ensuring that security in the face of server compromise is as in the exact checking case (since stored values remain unchanged). Thus we focus on analyzing online guessing attacks that seek to maximize their probability of success by exploiting the extra typo checks.

We prove a *free corrections theorem*. It states that there exists an optimal, fully secure typo-tolerant checker for any desired set of corrections. Consequently: (1) The optimal remote attack up to some query budget q is no more successful than the optimal attack against an exact checker and (2) No other checking scheme can improve the utility of typo corrections while maintaining no loss in security. The

key insight is that one can build a typo-tolerant checker that forgoes corrections in the rare cases when doing so will allow checking for multiple high-probability passwords.

Unfortunately, the optimal checker underlying the free corrections theorem must be based on exact knowledge of the password distribution—an assumption unlikely to be realizable in practice. We therefore explore the security of a number of practical typo-tolerant checkers, such as always correcting the top three typos, checking corrections only when they do not appear on a blacklist of common passwords, and a version of the optimal checker that uses the RockYou password leak [81] to estimate the password distribution. We then perform a number of simulations to show that these typo-tolerant checkers improve usability while remote guessing attacks improve negligibly, even in the worst case of attackers that somehow know the precise password distribution. For real attackers that estimate the distribution, our simulations suggest no improvement in online guessing attacks.

The contributions of this chapter are the following:

- We are the first to investigate the rate and nature of password typos made by users via measurement studies using Mechanical Turk and the production login infrastructure at Dropbox. Our work surfaces a small set of easily-correctable typos, such as capitalization errors, that alone prevent 3% of users from logging in during the period of study. Correcting these few typos could therefore non-negligibly boost user access to the Dropbox service.
- We introduce a formal framework for typo-tolerant password checkers and prove a free corrections theorem that establishes the existence, in theory, of an optimal typo-tolerant password checker that has no loss in security over exact checking.
- We introduce a number of practical typo-tolerant password checkers that are

compatible with existing password storage systems. Simulations show that these checkers can achieve no degradation in security yet still significantly improve login success rates.

Our focus is on web password ecosystems, but nothing about our techniques is unique to this setting. Our typo-tolerant checkers could easily be integrated in other settings such as logging into a desktop or laptop computer, though measurements are probably warranted to understand what are the best typo corrections for these other settings. We leave this to future work.

Immediate impact of our work. In the course of this research, our results prompted Dropbox to deploy a caps lock indicator on their website's password login interface. This indicator is like the one already available in Apple OS X, but appears in all browsers. Preliminary results suggest that it reduces caps lock errors by about 75%, and thus provides significant benefit. Unfortunately, it does not eliminate caps-lock errors nor assist with other sorts of common typos. So while our results in this chapter have already had a practical impact, we hope that they will also fuel further advances in the mitigation of password typos.

3.2 Background and Related Work

Password checking and threats. Traditional password-based authentication systems work as follows. A user chooses a username and a password at registration time. For subsequent logins, the user submits their username and password. Using some stored representation of the password (e.g., a salted hash), a password checking scheme determines whether the submitted password matches the registered

one. Login is allowed only if the equality check passes. A more formal treatment appears later in Section 3.5.

In terms of security, two main threats arise in the context of password checking systems. The first is online guessing attacks in which the attacker can submit guesses to the checking system via the standard interface. The attacker might target a particular login (a vertical attack), or try popular passwords against multiple accounts (a horizontal attack). Here the system can employ various countermeasures to mitigate online attacks, such as slowing down how quickly responses are returned, locking accounts after a certain number of queries per unit time, and using anomaly detection mechanisms to flag requests as unauthentic based on contextual information. The second main threat is leakage to attackers of password hash databases due to compromise of authentication systems or accidental data disclosure. These attackers can mount offline brute-force attacks in an attempt to crack the passwords. Our focus will be on online brute-force attacks as, looking ahead, our checkers will be compatible with existing password storage schemes and thus no alter security with respect to offline attacks. We discuss more in Section 3.5.

Typos in user-selected passwords. A number of prior measurement studies reveal the tendency of users to choose weak passwords [73, 67, 10] with highly skewed, heavy-headed distributions (i.e., a relatively small number of passwords are chosen by a large number of people). The most-stated reason is that ease-of-memorability guides users to simple, common passwords. While memorability is clearly a critical aspect of usability, users may also be reluctant to choose more complex passwords because entering them is difficult, error-prone, and slow. The problem may be exacerbated by various input device form factors, e.g., mobile

phone touch keyboards.

Few works have measured the difficulty of correctly entering user-chosen passwords. Keith et al. [74] measured the usability of user-selected passphrases in comparison to passwords for a cohort of 56 undergraduate students. They showed that 2.2% of entries of user-chosen passwords had a typo (defined by thresholding via Levenshtein distance), and the rate of typos roughly doubles for more complex passwords (at least length 7, one upper-case, one lower-case, one non-letter). A follow-up study by the same authors also revealed a typo rate of roughly 2% with another small corpus of students [75]. Their studies, being of small scale, may not generalize to other settings, and the authors do not analyze the types of errors subjects made.

Mazurek et al. [82] hypothesize that users may pick weaker passwords because they are simpler to type and that more complex passwords are harder to type. Via large-scale measurements of a university authentication system, they show that login errors are correlated with stronger passwords. However, they do not analyze the nature of errors, i.e., whether they were in fact typos, typing in the entirely wrong password, or some other problem.

Server-side hashing changes. In theory a secure sketch [83] could be used to correct some typos in the server side. However, the proven bounds for existing constructions are too weak to provide meaningful protection for our setting (in which entropy is quite low). More details are given in Appendix B.1. Mehler and Skiena [84] propose to allow controlled collisions in password hashing so that, with high probability, passwords with a transposition or substitution error hash to the same value. Both such approaches to typo-tolerant techniques are not backwards-compatible with existing password storage and also will degrade offline attack

security.

Typos in passphrase systems. Shay et al. [71] perform a study of system-generated passwords that are chosen uniformly for a user, chosen uniformly to be pronounceable, or chosen uniformly among CorrectHorseBatteryStaple-type passphrases [85] of various word lengths. They measure typos and investigate the correlation between password/passphrase length and typing errors, and investigate simple typo-tolerance strategies such as ignoring case completely and, for passphrases, combining words from a dictionary whose strings have large pairwise Damerau-Levenshtein distance [86, 87], which, in turn, enables correction by comparison with the dictionary. This latter suggestion is originally due to Bard [88]. Later, Jakobsson and Akavipat [89] suggest similar dictionary-checking-based error correction in what they call fastwords. In contrast to the above works, we focus on arbitrary user-chosen passwords, so these previous measurements and mechanisms unfortunately do not apply to our setting.

Typo-tolerant checking in industry. There have been several examples of major websites accepting slightly incorrect versions of user-chosen passwords. Facebook as early as 2011 accepted the correct password, the password with all letters' cases flipped, or the password with the first letter's case flipped (if it is indeed a letter) [76, 77]. These two modifications correspond to errors resulting from leaving the caps lock on (or off) or the tendency of (particularly) mobile phone keyboards to automatically capitalize the first entered character. Early password authentication mechanisms at Amazon allegedly ignored case and any characters beyond the eighth position due to a bug [90]. Users of Vanguard (an investment management company) reported that the answers to security questions could have typographical errors and still be accepted [79].

These companies faced significant backlash in the media and from some security professionals [79, 90, 77]. The assumption underlying the criticism seems to be that accepting any variant of a password will necessarily speed up online guessing attacks.

Open questions. To summarize, before our work there was no information available about the kinds of typos that burden users typing user-selected passwords and whether typo-tolerant password checking systems are achievable without degrading security. We seek to answer these questions here.

3.3 Understanding Typos Empirically

We start with experiments using Amazon Mechanical Turk (MTurk) [91] to measure the kinds of typos that people make when typing passwords. The goal of this preliminary measurement study is to discover the most frequent typos across a population for typical user-chosen passwords. We will follow on up these MTurk experiments with real user data using instrumentation of the Dropbox operational environment (discussed in Section 3.4).

Experiment design. MTurk allows custom-designed human-intelligence tasks (HITs) to be submitted to workers over the web. We created a password-typing HIT that asks a worker to type k passwords within a given time limit. Inside a HIT, every password needs to be typed within a conventional HTML password-type input box, i.e., each typed character shows up as a dot. Copy-paste functionality is disabled in the input boxes using the html "`onpaste=false oncopy=false`" option. This check can be circumvented by changing the browser settings, but

we recorded all key presses inside an input box and used this to help filter out copy-paste attempts. We did not find any circumvention in the collected data.

Our MTurk experiment design mainly aims at gathering data efficiently to identify common typos and trends. We note that the typos found in transcribing passwords in MTurk may not be truly representative of the typos users make when typing their own passwords. Performing a longitudinal study using MTurk where users retype their chosen password multiple times would be interesting, but it would be logistically complicated and would greatly slow the data collection rates while still not providing real ecological validity [92]. The reasons for experimenting in MTurk is that prospecting for common typos in a real operational environment, such as Dropbox's, would seem to require storing information about plaintext passwords in between logins, which could represent a significant security problem. Thus we adopt the two-phase investigative approach. First prospecting for common typos via MTurk and, given a list of such typos, presenting a measurement of real-world Dropbox user typos later in the chapter.

In our MTurk experiments we ask workers to type the passwords which are sourced from the RockYou password leak [81]. This data set is the largest plaintext password leak to date, with passwords from over 32 million users. It has been used widely for password-related studies and the distribution of passwords is similar to other leaks. The data set contains a number of passwords that may be objectionable to some people (e.g., many popular passwords are based on profanities from a wide number of languages). Instead of removing these passwords, which would bias the study, we used the MTurk mechanism of indicating that there may be vulgar content in the HITs. This restricts the HITs to be used only by adult workers as well as providing a warning to them about the potential for

objectionable content. We also removed all passwords of length greater than 25, as these are (by manual inspection) not user-selected passwords.

Amazon allows the HIT creator to specify the required qualification and location of the worker. We allowed workers with more than 10% acceptance rate[2] and that were located in countries whose official language is English.

None of the data we recorded contains personally identifying information. We nevertheless submitted our experiment designs to our institutional review board and received an IRB exemption.

3.3.1 Measured Typo Rates

We sampled 100,000 passwords randomly with replacement according to the empirical probability distribution of RockYou passwords of length 6 or more. (The length requirement matches the Dropbox policy for passwords, as discussed in the next section.) By sampling with replacement, this means we match the expected distribution of submitted passwords a web service might see across their entire user base (e.g., the password "123456" appears frequently). We split the sample into HITs, ensuring that none of the HITs contain more than 180 characters in total. (This approximately normalizes the amount of typing effort of a single HIT.) Each MTurk worker is given 300 seconds to type all the passwords in the HIT. The number of passwords in a HIT ranges between 16 to 22. To ensure a broad pool of users, a worker is only allowed to submit a maximum of 3 HITs. To impose this restriction, we used a third-party JavaScript function provided by a website called

[2]Acceptance rate in MTurk paradigm means the percentage of HITs, that the worker has submitted, have been accepted by the requester of the work. This is an eligibility filter provided by MTurk.

Unique Turker [93]. In addition to the submitted passwords, we collected the user agent string of the browser from which the worker submitted the job, and all the key presses (and their timestamps) inside the input boxes within the HIT.

A total of 4,362 workers participated in our study. Several passwords were not typed at all (e.g., the worker accidentally submitted the HIT before typing all the passwords), and sometimes the wrong password was entered (e.g., when prompted "123456" the user entered "password").

Sanitization. We sanitize the received data first, by removing the submissions where either no password was typed or typed passwords have a case-independent edit distance of five or more from the prompted password. This excluded 226 of the password samples. Here and throughout this section edit distance includes insertions, deletions, and substitutions each as unit cost.

Preliminary analysis of the remaining data revealed that a large fraction of errors were caused by accidental pressing of the caps-lock key. Looking at the data, it was clear that in many cases workers had caps lock on for a large number of contiguous entries. We therefore sanitized our data with a heuristic to figure out improper propagation of caps-lock errors across multiple entries. The details are given in Appendix B.2.

After sanitization, there were in total 4,364 incorrect submissions across 97,632 valid submissions (4.5%). There were 81,595 unique passwords among the sanitized submissions, and 5.5% of all unique passwords were mistyped at least once. Because we instrumented all key presses, we could see when users corrected entries before submission. An additional 8.2% of submissions were first incorrectly typed by the workers, but corrected before submission. In total, we found that 42% of the

workers made at least one typo across all their submissions, while 1.6% submitted more than four mistyped passwords in their submissions.

From now on our analyses are based on the submitted passwords unless otherwise specified. We include duplicate passwords in our analyses because they reflect the distribution of passwords a provider would see.

The data resulting from the MTurk measurements suggest that there is some correlation between typo likelihood and password complexity under various measures such as length and lexical diversity. As this is not our main focus we defer discussion to Appendix B.3.

3.3.2 The Nature of Typos

We now analyze the nature of typos made by the MTurk workers. First, we look at typos based on the edit distance between the mistyped password and the correct password. For 86% of incorrectly typed passwords, if we normalize the cases of alphabetic characters, the edit distance between the submission and the correct password was one. This suggests most password typos are relatively simple.

To obtain better clarity on the kinds of typos made, we analyze typographical errors on a representation of strings that accounts for the keys that must be pressed while entering it. This allows us in particular to highlight the role of capitalization errors due to shift and caps-lock mistakes during password entry.

The key-press representation of a string is defined as follows. First, recall that our standard alphabet includes upper- and lower-case letters, numbers, symbols, and the space character. We define a key-press alphabet that includes only the

keys on a standard US keyboard: lower-case letters, numbers, symbols that can be entered without shift (such as the period), the caps-lock key denoted by $\langle c \rangle$, and either shift key represented by $\langle s \rangle$. Then, we convert each password and submitted string from the MTurk study to a key-press string by replacing characters omitted from the key-press alphabet by appropriate combinations of $\langle s \rangle$ or $\langle c \rangle$ tokens and tokens for keys in the key-press alphabet. We heuristically assume any sequence of 3 or more capital letters was entered using caps lock and singleton or doubles using a shift key. So for example, the string "Password" would be converted to the string "$\langle s \rangle$-p-a-s-s-w-o-r-d" over our new alphabet, and likewise "ABC12!@" would be converted to "$\langle c \rangle$-a-b-c-$\langle c \rangle$-1-2-3-$\langle s \rangle$-1-$\langle s \rangle$-2". As seen in this last example, in our conversion we insert a $\langle s \rangle$ token for each character modified by it (despite the fact that the user may hold it down for the duration).

To gain insight into what kinds of typos the workers made, we constructed a confusion matrix in which rows represent the true keys that should have been pressed, and columns represent the keys that were actually pressed. We filled this matrix in the following way. For each pair of prompted password and submitted string, we find an optimal alignment of the corresponding key presses that minimizes the total cost of the edit operations. We can extract an optimal alignment in the process of computing the minimum edit distance using a dynamic programming algorithm approach proposed in [94]. We then counted the frequency of $c \to c'$ pairs, where c is the key in the given password and c' is the key which is typed. We allowed c to take on a placeholder value [ins] signifying when c' was inserted into a password, and c' to take on a placeholder value [del] to denote that c was deleted from a password. We omitted the case $c = c'$ from our tabulation, as it represents no typographical error.

For example, consider if the prompted password was the string "Password", (or "$\langle s \rangle$-p-a-s-s-w-o-r-d" in key-press representation) and the study participant submitted the string "passw0rd1" (or "p-a-s-s-w-0-r-d-1" in key-press representation). Our algorithm would increment the counts for $\langle s \rangle \to$ [del], $o \to 0$, and [ins] $\to 1$. The corresponding typos are: forgetting $\langle s \rangle$ to capitalize the first letter, changing an 'o' to '0', and adding a (spurious) '1' at the end.

The histogram resulting from doing this for all submitted passwords is shown as a heatmap in Figure 3.1. Darker colors signify higher counts. The keys are sorted according to a standard US keyboard layout.

Several common typographical errors stand out:

- *Insertion and deletion of shift and caps-lock keys*: In the right bottom corner appears a dark patch of 3×3 squares. This reflects the frequency of erroneous use or lack of use of shift and caps lock—equivalently, incorrect insertion or deletion of the $\langle s \rangle$ and $\langle c \rangle$ tokens. These typos will switch the case of the password if it contains English letters as well as changing the shift status of digits and symbols (e.g., $4 \to \$$).
- *Keyboard proximity errors*: The slightly darker cells near the diagonal represent typos due to mistakenly pressing a neighboring key to the left or right of the intended key. We found more generally that there are a significant number of typos for which a key is replaced by an adjacent one (left, right, above, or below). We collectively refer to these as proximity errors.
- *Number-to-number errors*: We see a square cluster of moderately high-frequency errors in the top left that represent digit-to-digit typos. Some of these are proximity errors, but many such errors confuse widely separated numbers, e.g., $3 \to 9$.

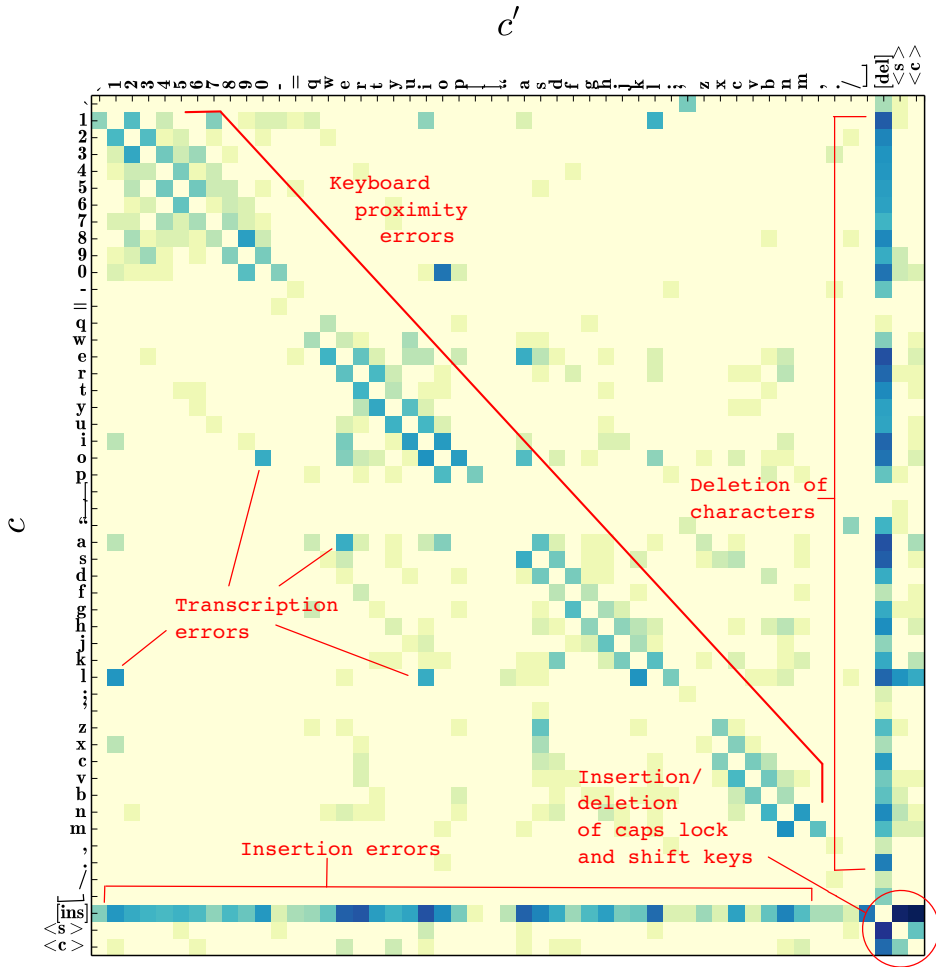


Figure 3.1: Heatmap showing the counts of edits that arose in computing edit distance from the key-press sequence of the submitted passwords to the key-press sequence of the prompted passwords. The color in row c and column c' indicates how often the edit $c \to c'$ was observed across all distance calculations. The darker the color the higher the count. Labels [ins] and [del] denote insertion (character mistakenly inserted) and deletion (failure to type a character). Tokens $\_$, $\langle s \rangle$, and $\langle c \rangle$ respectively denote the and space-bar, shift, and caps lock.

- *Insertions and deletions*: There are throughout a large number of insertion (third row from the bottom) and deletion (third column from the right) errors. Deletions are slightly more common.

- *Transcription errors*: The heatmap has sporadic dark cells, including (l,1), (o,0), (0,o), (l,i). These represent transcription errors due to a worker confusing similar-looking characters. We presume that the prevalence of read-

ing errors are an artifact of the experiment design, and will be less frequent for entry of memorized passwords. Nevertheless, such errors could arise for users that write down their passwords to remember them.

Our analysis suggests that a large fraction of common typos fall into a few classes. A subset of these are what we refer to as "easily correctable," as we discuss shortly.

3.3.3 Touchscreen Keyboards

We performed a smaller, but similar, study in which workers were required to use touchscreen keyboards. The hypothesis here is that the distribution of typos may differ due to keyboard type. We submitted 24,000 passwords drawn from RockYou across 1,987 HITs using the same methodology of approximately normalizing effort by restricting total character counts to be less than 110. Workers were given 300 seconds to perform a HIT. We restricted workers to using touchscreen keyboards by checking the `user-agent` string of the worker's browser.

Unlike the desktop user experiment earlier in this section, we did not need to adjust for the caps lock propagation error on touchscreen devices. This was because of the fact that in touch screen devices the caps lock key is auto reset every time the focus shifts from one input field to the other. We performed an analysis that was otherwise similar to the analysis used above for the general MTurk experiment. To calculate proximity errors, we used the Android keyboard layout, which we believe is a sufficiently good proxy for all touch screen keyboards.

The probability of a typo here was 9.0%, an increase over the 4.5% for unrestricted workers. We compare the types of typos across the two data sets below.

Typo type	Corrector	% of typos	
		Any	Mobile
Case of all letters flipped	swc-all	10.9	8.3
Case of first letter flipped	swc-first	4.5	4.7
Added extra character to end	rm-last	4.6	0.9
Added extra character to front	rm-first	1.3	0.5
Missed shift for symbol at end	n2s-last	0.2	0.1
Proximity errors	n/a	21.8	29.6
Transcription errors	n/a	3.0	3.3
Other errors	n/a	53.6	52.7

Figure 3.2: The top categories of typos observed in our MTurk experiments. The "Corrector" column identifies an (easily applied) function that corrects the typo. The "Any" column is percentage of typos by category for the initial MTurk study in which workers could have used any browser. Of 97,632 passwords drawn from RockYou, 4,364 were mistyped. The "Mobile" column is the same for the 23,098 submitted passwords collected from devices with mobile browsers. Of these, 2,075 had a typo.

3.3.4 Easily-Correctable Typo Classes and Correctors

Using all the data above we manually enumerate a set of common typo types, or classes. The resulting classes are detailed in Figure 3.2, and shown for both the first general MTurk experiment and the touchscreen-restricted experiment. The column labeled "Corrector" identifies the function that can be used to correct the corresponding typos: swc-all switches the case of all letters in a password, swc-first switches the case of the first letter, rm-last removes the last character, rm-first removes the first character, and n2s-last changes the last character to its equivalent character under the shift-key modifier (e.g., '1' becomes '!', 'a' becomes 'A', etc.). The correctors mentioned above are mutually exclusive, that is, any two correctors, when applied to an input password of length larger than one, will produce two different passwords (assuming at least one of the correctors is applicable).

As can be seen, the distribution of typos is non-uniform. A few typo classes account for a large proportion of mistakes made. Caps-lock errors alone represent

9.2% of all mistakes made in our general MTurk experiments, and proximity errors for another 21.8% of all mistakes. For mobile, we see a proportionally larger number of keyboard proximity typos.

If a class of typo has a uniquely determined associated corrector, we refer to it as *easily correctable*. The typo that produces a flipped case in the first letter is an example: The corresponding corrector just flips the case of the first letter. Not all easily correctable typos have involutory correctors (the typo and corrector are the same function): consider the case of adding a character to the end of a password which is corrected by removing a character.

In contrast to easily correctable typos, a proximity error is hard to correct. Given a password with a proximity error, correction would require identification of the erroneous character as well as identification of the nearby character that was the original, true one. Thus the space of possible correctors for a proximity error is generally large. As we shall see later, both security and performance are adversely impacted by searching large spaces of correctors.

Our exploration culminates in the following two key results: *(1) Some typos are significantly more common than others* and *(2) Many common typos are easily correctable*. In the next section, we report on experiments at Dropbox that verify that common, easily correctable typos arise frequently in practice.

3.4 Experiments at Dropbox

Our Mechanical Turk experiments in the last section show that there exists a small set of frequently observed typos. Those experiments, which asked users to

type in passwords provided to them, may not simulate the kinds of typos users make when using their own passwords. We therefore turn to investigating typos in the production password authentication environment used at Dropbox. We will also assess the impact of typos on user experience. We emphasize that our experiments here did not change the effective login checks at Dropbox, but only recorded information about the frequency of typos.

The Dropbox authentication system. Dropbox is a file hosting service for consumers and enterprises with hundreds of millions of users. Each user must select a password during registration. Dropbox uses zxcvbn [95], a password strength estimator, to guide the user in choosing a strong password. The system requires that users choose a password of at least six characters, but it does not explicitly forbid users from choosing passwords that are considered to be weak by zxcvbn. Passwords are submitted over a standard HTTPS POST interface when logging in via the website or from within one of the native Dropbox applications. We call the submission of a password by a user a *password submission*. If the password is accepted by the Dropbox server, we call it a successful password submission, otherwise it is called a failed password submission. On a failed password submission, the user may resubmit his/her password. A *login attempt* is a sequence of password submissions by a user that either culminates in a successful login, in which case the login attempt is considered successful, or accumulates login failures until the study ends. If the user does not succeed in logging in during the scope of our study, we consider her sequence of password submissions to be a failed login attempt.

Dropbox, like most modern web companies, uses a number of fraud detection mechanisms in order to filter out spurious login attempts even before checking the password. An example of such fraud detection mechanisms is to refuse login

attempts from IP addresses that appear on a blacklist for known bots. While some spurious login submissions may make it through these filtering mechanisms, we assume for simplicity below that our instrumentation is only monitoring legitimate login attempts. Note that this is a conservative assumption: if the data we collected contains illegitimate login attempts, then the true rate of correctable typos for legitimate users would be even higher. Our security analyses (Section 4.5) will not make such assumptions.

Instrumentation. We modified the Dropbox password checking code to perform additional checks on all legitimate login attempts on the web interface. This provided a vast amount of data, and it eliminated biases that could arise from selecting some small percentage of accounts. This also made visible multiple password submissions from a single user, which was necessary for timing re-tries.

During the period of measurement, every password submission was processed as follows. If the password check passed, do nothing. Otherwise, if it failed, apply one or more typo corrections from some predefined corrector function set $\mathcal{C} = \{f_1, f_2, \ldots, f_c\}$ where corrector functions were defined in the last section. We used slightly different sets of correctors in different experiments, as discussed below. One or more of the corrected version(s) of the password are checked. For failed login attempts, a log entry was generated that contained a time stamp, whether login would have been successful with a correction of the password, the type of correction f_i that was successful (if applicable), and the user agent string.

We emphasize that in our experiments login is *not* allowed based on the corrected passwords. We did not modify Dropbox's effective login checks; we only

collected the data needed to evaluate whether doing so would be beneficial.

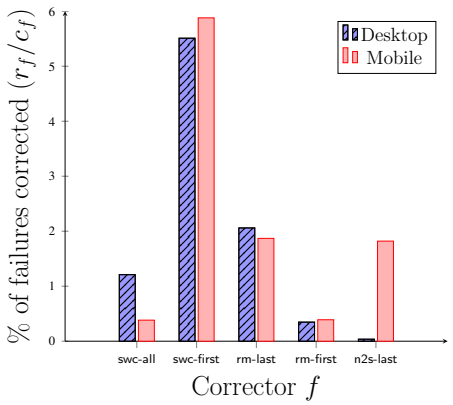
Typos and login failure rates. In an initial experiment we set out to measure the incidence rate of the top five corrections seen in the MTurk study of Section 4.6.1. Thus for this experiment the set of corrector functions is $\mathcal{C}_{\text{top5}} = \{\mathsf{swc\text{-}all}, \mathsf{swc\text{-}first}, \mathsf{rm\text{-}last}, \mathsf{rm\text{-}first}, \mathsf{n2s\text{-}last}\}$. For each instrumented failed password submission, one correction from $\mathcal{C}_{\text{top5}}$ was chosen uniformly at random and applied to the submitted password. The reason is that, in the current implementation, only sequential code is easily supported, and the password hashing scheme used at Dropbox is (by design) slow to compute. It was unclear a priori exactly what overhead the additional checks would have on Dropbox infrastructure, and so we conservatively only performed one additional check at a time. The success of this initial experiment suggested the performance impact was low, and later experiments applied multiple corrections (see below). We collected information over a 24-hour period.

We cannot report on the exact number of login attempts during this period, as this is considered confidential information by Dropbox. We will therefore report only rates of success and failure. In the following, we let c_f denote the number of times a corrector f was applied to an incorrect password during an experiment. We let r_f be the number of times f successfully corrected an incorrect password during the experiment. The ratio r_f/c_f gives the percentage of login failures correctable by f.

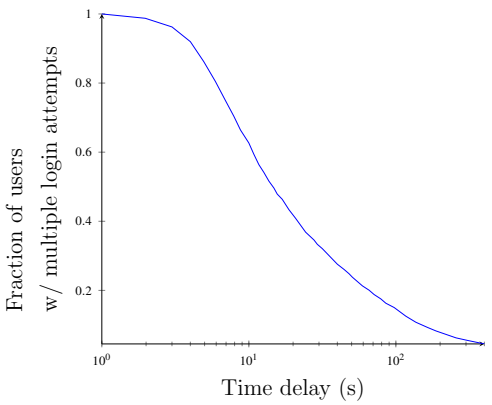
Figure 3.3 reports the measured ratios r_f/c_f for each corrector in $\mathcal{C}_{\text{top5}}$ in during the 24-hour period. This reveals that 9.3% of failures are due to typos correctable by $\mathcal{C}_{\text{top5}}$, suggesting that typos indeed account for a significant number of failed

Corrected by (f)	r_f/c_f (%)
swc-all	1.13
swc-first	5.56
rm-last	2.05
rm-first	0.35
n2s-last	0.21
$\mathcal{C}_{\text{top5}}$	9.30

Figure 3.3: The fraction of failed logins correctable by $\mathcal{C}_{\text{top5}}$ in a 24-hour study at Dropbox.



(a) Performance of $\mathcal{C}_{\text{top5}}$ on mobile versus desktop. For each corrector in $\mathcal{C}_{\text{top5}}$ we plot the fraction of failures for each platform correctable by the corrector.



(b) CDF of time delay (in seconds) between the first failed login due to a typo and first successful login. Included are only users that had a failed login attempt and later a successful one.

Figure 3.4: Results from the data collected by instrumenting Dropbox account.

(legitimate) password submissions[3]. By correction type, we see that the most common correction (switching the case of the first character) accounts for 60% of these, and the first three (switching the case of all characters, just the first character, dropping the last character) account for over 90% of these. Apparently capitalization errors are a significant source of errors, which provides evidence for why Facebook accepts these typos.

Some disparity with the MTurk results is apparent. While the top three of

[3]We can add the fractions of typos because our correctors are mutually exclusive.

these five correctors are the same, the ordering is distinct, with caps-lock errors proportionally higher in MTurk then here. We believe this is due to the MTurk experiment design, and that the Dropbox numbers more accurately reflect rates in operational environments.

While collecting this data, we recorded the user agent for all password submissions, so we were able to analyze the performance of typo correction on mobile platforms versus desktop platforms. We found that the estimated correction rate for mobile was slightly higher at 10.5%, compared to 9.3% for desktop (calculated here with the denominator being the number of rejected password submissions for mobile and desktop, respectively). We show, in Figure 3.4a, the estimated correction rates for each user agent broken down by corrector function. We see that n2s-last is a significantly more effective correction on mobile, which may be because mobile keyboards require switching to an alternate keyboard to reveal symbols. We also see that swc-all is a more effective correction on desktop, most likely because it's easier to leave caps lock enabled on conventional keyboards.[4] This dichotomy suggests the potential merit of applying different correction policies on the server based on the user agent. We leave the further analysis of this for future work.

Utility of the top three corrections. We perform a second study that restricts attention to just the overall top three correctors $\mathcal{C}_{\text{top3}} = \{\text{swc-all}, \text{swc-first}, \text{rm-last}\}$ observed in the previous study (and, in turn, the MTurk experiments). For this experiment, the instrumentation applied all three correctors to any password that failed to exactly match the registered password. So, now c_f is the number of failed login attempts for every $f \in \mathcal{C}_{\text{top3}}$. As before, we recorded data for 24 hours.

[4]On Android devices, enabling caps lock requires pressing and holding the shift button, and on iPhone devices one has to double press the shift button to enable caps lock.

We additionally recorded the time duration for a login attempt to succeed. That is the time lag between the first failed submission and the first successful submission by each user in this 24-hour period. (Because Dropbox uses session cookies most users typically need to successfully login only once per 24-hour period.) This allowed us to quantify the time delay between failures and successes, a measure of how much utility is lost due to usability issues such as typos.

As we would expect, the success rate of corrections closely matched the results of the previous 24-hour experiment. Specifically, typos correctable by $\mathcal{C}_{\text{top3}}$ accounted for 9% of failed password submissions. This also attests the stability of these percentages over time.

We show in Figure 3.4b a CDF of the delay in logging in over all users who eventually succeeded at logging in (within the 24-hour period). Note that some small fraction of users did not log in for a very long time, suggesting they gave up and came back hours later. Even so, almost 20% of users that experienced a failed login would have been logged in a minute earlier should typo-tolerant checking have been enabled. Aggregated across all failed login attempts, typo-tolerance here would have *increased logged in time by several person-months just for this 24-hour experiment*. This represents a significant impact on user experience and a clear pain point for companies keen on making it easy for their users to log in.

In aggregate, of all users who attempted to log into Dropbox within the 24-hour measurement period, we discovered that 3% were turned away even though at least one of their submitted passwords was correctable by one of the correctors in $\mathcal{C}_{\text{top3}}$. This also represents a significant impact on user experience, with users being prevented from using the service.

3.5 Typo-tolerant Checking Schemes

In previous sections, we saw that typos account for a large fraction of login failures and that a simple set of typo corrector functions could significantly improve user experience. A natural follow-on question is whether we can achieve typo-tolerance in password authentication systems without a significant security loss. We address that question here.

We will show, by introducing what we call the "free corrections theorem," that for all natural settings there exist typo-tolerant checking schemes that correct typos with *no security loss* relative to exact checking for optimal attackers that (unrealistically) have exact knowledge of the distribution of passwords. We will also specify the optimality of the scheme underlying this theorem, i.e., showing that it achieves the maximum utility possible with no security loss.

We will define the notion of a "natural" setting formally below. Intuitively, it corresponds to the highly non-uniform, sparse (in the space of all strings) passwords chosen in practice. The schemes we analyze formally are not readily applied as is in practice because, among other things, they require exact knowledge of password and typo distributions. Nevertheless, combing our measurement studies with a theoretical perspective guides us towards the design of several concrete typo-tolerant checking schemes for which we give empirical security estimates in Section 4.5.

3.5.1 Password and Typo Settings

Let $\mathcal{S}$ be a set of all possible strings that could be chosen as passwords, e.g., ASCII strings up to some maximum length. We associate to $\mathcal{S}$ a distribution p that models the probability of user selection of passwords; thus $p(w)$ is the probability that some user selects a given string $w \in \mathcal{S}$ as a password. We let $\mathcal{P} \subseteq \mathcal{S}$ be the set of possible passwords, which is formally just the support of p. We write $p(P)$ to denote the aggregate probability on a set $P \subseteq \mathcal{S}$ of strings. Following prior work (c.f., [7]), this model assumes for simplicity that the distribution of passwords is independent of the user selecting them, and that passwords are independently drawn from p.

A key feature of our formalization approach is that we do not appeal to a specific lexicographic notion of distance (e.g., Levenshtein distance) to model typos. Instead, we directly model typos as probabilistic changes to strings. Specifically, let $\tau_w(\tilde{w})$ denote the probability that upon authenticating, a user with password w types the string $\tilde{w}$. Thus τ is a family of distributions over $\mathcal{S}$, one distribution for each $w \in \mathcal{P}$. If $\tilde{w} \neq w$ then $\tilde{w}$ is a typo; $\tau_w(w)$ is the probability that the user makes no typo. Note that $\tilde{w}$ may or may not itself be a password possibly chosen by a user, i.e., it may not be in $\mathcal{P}$. We say that $\tilde{w}$ is a *neighbor* of w if $\tau_w(\tilde{w}) > 0$.

For all $w \in \mathcal{P}$, then, $\tau_w(\cdot)$ defines a probability space over $\mathcal{S}$. That is, $\tau_w(\tilde{w}) \in [0, 1]$ for any $\tilde{w}$ and $\sum_{\tilde{w} \in \mathcal{S}} \tau_w(\tilde{w}) = 1$. In practice, generally $\tau_w(w) > 0$, i.e., users will sometimes enter passwords correctly. Also, it will most often be the case that $\tau_w(\tilde{w}) \neq \tau_{\tilde{w}}(w)$ for $w \neq \tilde{w}$. For example, a user may mistype her password $w =$ "unlockme1" as $\tilde{w} =$"unlockme" as a result of accidentally dropping the last 1, while a user whose password is $\tilde{w} =$"unlockme" is less likely to type a 1 at the end of his password.

In our model we assume that typos depend only on a user's password w and not, for example, on the user that typed them, the time of day, or other factors. As we will see, this assumption simplifies operationalization of typo tolerance models. As one example, modeling individual users' typo habits would require a server to record the user's typo history. While higher-accuracy correction for the user might then be possible, this feature would, of course, result in a more complex system. It could also leak password information: recording the fact that a user fails to capitalize the first character in her password leaks the fact that character is a letter. From now on, a password and typo setting, or simply *setting*, is a pair (p, τ).

3.5.2 Password checkers

A password checker scheme consists of two algorithms:

- Reg is a randomized password registration algorithm. It takes as input a password w and outputs a string s that may, for example, be the output of a password hashing scheme like scrypt. These are randomized since one must choose a random salt value for each registration.
- Chk is a (possibly randomized) password verification algorithm. It takes as input a string $\tilde{w}$ and a stored string s, and outputs a Boolean value, either true or false.

In a modern, real-world service such as Dropbox, Chk is one input in a complex authentication system that combines multiple contextual, potentially probabilistic signals to make an authentication decision. A typo-tolerant checker could return a

probabilistic estimate and/or combine with other contextual signals, but we focus our analysis only on deterministic checkers. Our techniques extend in natural ways to confidence values (e.g., by returning an estimate of $\tau_w(\tilde{w})$). In such a scenario, the security impact of a typo-tolerant Chk will be even lower. We also consider only *complete* checkers, meaning that for all w, $\mathsf{Chk}(w, \mathsf{Reg}(w)) \Rightarrow \mathsf{true}$.

An *exact checker* is one which never outputs true if $\tilde{w} \neq w$. In practice of course, exact checkers actually have a non-zero, but cryptographically small probability of false acceptance (for typical hash-function-based checkers, this small probability is equal to the probability of having found a collision in the hash function). We will throughout ignore this false acceptance probability. We will use ExChk to denote some secure exact checker, and assume the existence of one compatible with all password settings of interest.

Typo-tolerant checkers. We will focus our attention on building typo-tolerant checkers that *relax* the checks made by an existing exact checker construction. Let $\mathsf{Reg}, \mathsf{ExChk}$ be the algorithms of an exact checker. Then an associated relaxed checker has the same registration algorithm, but a different checking algorithm $\mathsf{Chk} \neq \mathsf{ExChk}$. Specifically, our approach will be to design relaxed checkers that enumerate some number of strings as candidates for the password and checks each with an exact checker.[5] The *ball* of a submitted string $\tilde{w}$ is the set $B(\tilde{w}) \subseteq \mathcal{S}$ of checked strings.

If balls are well constructed, the hope is that it often happens that when the user makes a typo, the true password w lies in the ball around the user submitted string $\tilde{w}$, and thus the typo can be corrected.

[5] We note that this can be viewed simply as the standard brute-force construction of an error correction code from an error detection code.

Relaxing an exact checker is a desirable approach to typo-tolerance for two main reasons. The first is *legacy compatibility*. Modifying a system to become typo-tolerant just requires deploying a new checking algorithm that works with previously registered passwords. For example, registration may use a password hashing scheme like scrypt [96] or argon2 [97], or a password onion construction that combines password hashing with an off-system crypto service [80].

Second, relaxed checking offers *no security loss against offline, brute-force attacks* when the exact checker has, underlying it, a secure hash function. A compromise of the system or leak of the password hash database gives an attacker the registered string s, just as in the case of the exact checking system. When s is computed by applying a secure password hashing algorithm (e.g., [35, 96, 97]), an offline attacker's goal is to perform brute-force attacks to recover a password. Here one may worry that the attacker's goal is easier as it requires simply inverting s to a point that is in the ball of the target password, but for secure hash functions nothing will be revealed about the target password by s until the target password is found exactly. Thus, for a given user account, either an adversary: (1) Cracks a password hash and submits the correct password, in which case she obtains no advantage in an online attack from typo-tolerance or (2) Fails to crack a password hash, in which case she gains no benefit from her offline attack in mounting an online attack. Of course, should the typo-tolerant Chk algorithm be very complex to implement, it might increase the likelihood of software implementation vulnerabilities. For this reason, we consider simple-to-implement relaxed checkers.

Security degradation in a relaxed checker may still arise in *online* attacks. A poorly conceived relaxed checking system could diminish system security against remote brute-force guessing attacks. We will investigate this issue in detail below.

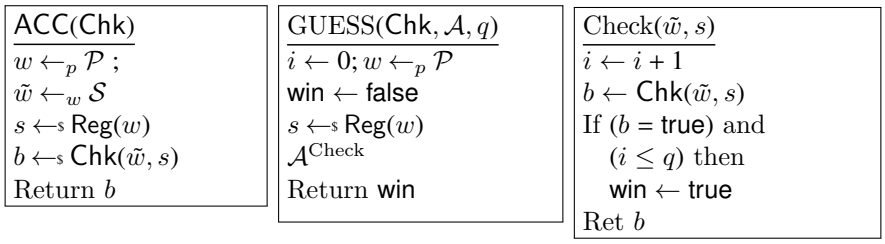


Figure 3.5: (**Left**) Experiment for defining acceptance utility for a checking scheme Reg, Chk. (**Right**) Security game for online guessing attacks against a checking scheme Reg, Chk in which $\mathcal{A}$ may make q calls to its oracle Check. Both experiments are implicitly parameterized by a password and typo setting (p, τ).

Before doing so, we note that relaxing an exact checker does circumscribe the space of possible checker designs. In particular, the size of a feasibly searchable ball $B(\tilde{w})$ is necessarily somewhat small: ExChk is designed to be computationally expensive to thwart offline brute-force guessing attacks, and relaxed checking involves running it for each string in $B(\tilde{w})$. Our measurement results in the prior sections show that even for balls of size three or four, however, significant utility improvements are possible.

Acceptance utility. We measure utility of a relaxed checker by the probability that the checker outputs true for entered passwords even when the submitted string is a typo of the true password. Formally, the acceptance utility is defined to be $\mathsf{Utility(Chk)} = \mathsf{Pr[ACC(Chk)} \Rightarrow \mathsf{true]}$, where the event captures the probability that the experiment of Figure 3.5 outputs true. There $\leftarrow_p$ means sampling from the set according to p, and $\leftarrow_w$ means sampling from the set according to τ_w. The game is (implicitly) parameterized by the registration algorithms and the distribution pair (p, τ), and models a user's choice of password and first attempt to enter it.

The acceptance utility of an exact checker is $\mathsf{Utility(ExChk)} = \mathrm{E}[\,\tau_w(w)\,]$ where the expectation is over $w \leftarrow_p \mathcal{P}$. For any non-trivial distribution τ, i.e., assuming a non-zero typo probability for some password, $\mathsf{Utility(ExChk)} < 1$.

3.5.3 Security definitions

As discussed above, since we focus on relaxed checkers, attacks due to compromise of an authentication server are not affected by a shift to typo-tolerance. The critical question is the effect of typo-tolerance on online guessing attacks.

Let us precisely define the notion of an online attack. In Figure 3.5 we give a simple guessing game played between an adversary $\mathcal{A}$ and a checker. The game GUESS is implicitly parameterized by p and the checker $\mathsf{Reg}, \mathsf{Chk}$. The success rate of the adversary $\mathcal{A}$ in guessing the password is measured as $\mathsf{AdvChk}, \mathcal{A}, q) = \mathsf{Pr}[\text{GUESS}(\mathsf{Chk}, \mathcal{A}, q) \Rightarrow \mathsf{true}]$. This security game models a vertical attack, where the attacker tries to compromise a randomly chosen user account; changing this security game to model horizontal attacks is straightforward and our results extend to this setting as well.

Measuring security by this definition is quite conservative because it ignores the many countermeasures used in practice to thwart online guessing attacks. Most companies implement anomaly detection mechanisms that would, for example, block attackers that query too quickly, that use a known cracking tool or password leak to generate guesses, or that mount attacks from suspicious-looking IP addresses (those in the wrong country or on a botnet blacklist). Thus our evaluations here and in the remainder of the chapter should be considered pessimistic upper bounds on true success rates.

Optimal and greedy attacks. Let $w_1, w_2, \ldots$ be a non-increasing order on passwords by probability, i.e., $p(w_1) \geq p(w_2) \geq p(w_3) \geq \ldots$. If a checker ExChk is exact, then $\mathsf{Adv}(\mathsf{ExChk}, \mathcal{A}, q) \leq \lambda_q$ for any $\mathcal{A}$ making at most q queries and where $\lambda_q = \sum_{i=1}^{q} p(w_i)$. Often λ_q is called the q-success rate. It was first defined as a

measure of the unpredictability of a password distribution by Boztas [98].

Now consider a relaxed, deterministic checker. Let $B(\tilde{w})$ be the ball of a string $\tilde{w}$ for a checker Chk, which is the set of all passwords for which Chk will accept $\tilde{w}$. In submitting a guess $\tilde{w}$, an attacker induces checking on all of the strings in $B(\tilde{w})$. The adversary knows the design of the checker and so too can determine what ball will be associated with any given string submitted to the checking oracle.

Define λ_q^{fuzzy} to be the maximum guessing success probability of any adversary, namely

$$\lambda_q^{\text{fuzzy}} = \max_{\mathcal{A}} \ \mathsf{Adv}(\mathsf{Chk}, \mathcal{A}, q) \ .$$

The dependence of λ_q^{fuzzy} on Chk is left implicit in our notation but will be clear from context later. For $q = 1$, $\lambda_1^{\text{fuzzy}} = \mathsf{argmax}_{\tilde{w} \in \mathcal{P}}\, p(B(\tilde{w}))$. An optimal attacker simply guesses the password $\tilde{w}$ whose ball has the highest aggregate probability. This guessing strategy is analogous, in an exact-checking setting, simply to guessing the most probable password w_1. We observe that λ_1^{fuzzy} as defined here coincides conceptually with the fuzzy min-entropy notion of Fuller et al. [99], hence the fuzzy superscript in λ_q^{fuzzy}.

It turns out that implementing an optimal attack is, in general, NP-hard: finding the optimal set of queries is an instance of the weighted max cover problem. The formal reduction is shown in Appendix B.5. This is good news for security: it means that attackers cannot in general compute the optimal queries to make. That said, there exists a conceptually simple greedy algorithm that we now give.

Consider the following greedy adversary $\mathcal{A}^*$. At each step, it guesses the password $\tilde{w}$ whose residual ball $B(\tilde{w})$ has the highest aggregate probability. This ball is the one that maximizes $p(B(\tilde{w}) \cap P)$, where P is the set of residual passwords,

those not yet checked by Chk as a result of previous adversarial queries.

More precisely, $\mathcal{A}^*$ does the following. Initialize a set $P = \mathcal{P}$ of possible passwords. Then repeat the following q times. Guess a string $\tilde{w}$ that maximizes $p(B(\tilde{w}) \cap P)$. If the query succeeds, then the game is won; otherwise set $P \leftarrow P \setminus B(\tilde{w})$ and repeat. Let $\lambda_q^{\text{greedy}} = \mathsf{Adv}(\mathsf{Chk}, \mathcal{A}^*, q)$. As by the reduction of this problem to max cover we can claim using the classic result [100], that $\lambda_q^{\text{greedy}} \geq (1-1/e)\lambda_q^{\text{fuzzy}}$. Furthermore, Feige [101] has shown that this performance is indeed optimal, and no polynomial time approximation algorithm outperforms the performance of greedy.

All this gives us a way to measure security of a relaxed checker given an estimate of the password distribution p: simply compute $\lambda_q^{\text{greedy}}$ for the threshold q on online queries relevant to ones' system. This gives one, in all likelihood, the best attacker one will face in practice. One can also obtain a worst-case bound of λ_q^{fuzzy} by the formula above.

Computing even $\lambda_q^{\text{greedy}}$ in the most obvious way—a naive execution of $\mathcal{A}^*$—has time complexity on the order of $|\mathcal{S}|$ times the average ball size, and so will generally itself be intractable. We propose an alternative approach to restrict the search space and allow one to compute $\lambda_q^{\text{greedy}}$ for relevant q efficiently. The details appear in Appendix B.5.

Security loss. The above gives us a way to bound absolute security, but our concern will primarily be the gap between the security of today's current practice of exact checkers and the security of relaxed versions of them. This clearly depends on the password distribution and typo setting. We measure loss relative to the greedy attacker by $\Delta_q^{\text{greedy}} = \lambda_q^{\text{greedy}} - \lambda_q$ and worst-case loss by the difference

$\Delta_q = \lambda_q^{\text{fuzzy}} - \lambda_q$. As $\lambda_q^{\text{fuzzy}} \leq \frac{e}{e-1} \cdot \lambda_q^{\text{greedy}}$, we can bound $\Delta_q \leq \frac{e}{e-1} \cdot \Delta_q^{\text{greedy}} + \frac{\lambda_q}{e-1} \approx 1.582\,\Delta_q^{\text{greedy}} + 0.582\lambda_q$.

By definition, $\lambda_q^{\text{fuzzy}} \geq \lambda_q$, meaning that $\Delta_q \in [0, 1)$. Moreover it holds that $\lambda_q^{\text{fuzzy}} \leq c\lambda_q$ for any tolerant checker that checks at most c strings for any input string $\tilde{w}$, i.e., $|B(\tilde{w})| \leq c$ for all $\tilde{w}$. This inequality is in fact an equality for some settings. Consider when p is uniform over $\mathcal{S}$ and that Chk is such that $|B(\tilde{w})| = c$ for all $\tilde{w}$. Then moving to the typo-tolerant checker will increase the probability of success of the optimal online brute-force attacker by a factor of c. Formally, $\lambda_q^{\text{fuzzy}} = c\lambda_q$ whenever $q \leq |\mathcal{S}|/c$.

This example seems to underlie the intuition for why typo-tolerance has been criticized as a security issue [77, 79]. Indeed, it is tempting to conclude that typo tolerance will *always* result in a factor c decrease in security. But this conclusion is too hasty: p is not uniform in reality and, in particular, passwords with high mass are sparse in the universe $\mathcal{S}$. Sparsity matters since a high λ_q^{fuzzy} depends intimately on finding strings whose balls include many passwords with high mass under p. In fact, we show next that for most natural settings one can actually obtain no security loss relative to an exact checker.

3.5.4 Free corrections theorem

We would ideally like to have typo-tolerant checkers that enjoy *free corrections*. This means that its security is equivalent to the security of an exact checker and so $\Delta_q = 0$ for any reasonable q. It is easy to come up with artificial distributions which admit free corrections of all typos. Specifically, a distribution for which no password's neighbor is in the ball of another password.

Unfortunately, just as the uniform setting, the dense setting discussed above is artificial, this completely sparse setting is also not realistic. For example, in the RockYou password leak and taking $\mathcal{C}_{\text{top5}}$ as the set of corrections to apply, one has significant overlap even among the top 50 passwords. We therefore ask: for the password distributions seen in practice, can one achieve checkers with free corrections? The answer is yes.

An optimal relaxed checker. We first give a construction of a relaxed checker that achieves free corrections for any given set of corrector functions $\mathcal{C} = \{f_0, f_1, \ldots, f_c\}$ one wants to consider, where $f_0(\tilde{w}) = \tilde{w}$ is the identity function. It achieves best-possible acceptance utility and no security loss relative to the best possible attack, assuming the checker has exact knowledge of the distribution pair (p,τ).

Fix some query budget q and recall that $p(w_q)$ is the probability mass of the q^{th} most probable password. Then let OpChk be the typo-tolerant checker that works as follows. Upon input $\tilde{w}$, generate a list of candidate typo corrections $\hat{B}(\tilde{w}) = \{w' \mid w' \leftarrow f_i(\tilde{w}),\ f_i \in \mathcal{C},\ \text{and } p(w') \cdot \tau_{w'}(\tilde{w}) > 0\}$. After this OpChk solves the following optimization problem to compute the set B,

$$\begin{aligned} \underset{B \subseteq \hat{B}(\tilde{w})}{\text{maximize}} \quad & \sum_{w' \in B} p(w') \cdot \tau_{w'}(\tilde{w}) && \text{/* Utility */} \\ \text{subject to} \quad & p(\tilde{w}) > 0 \Rightarrow \tilde{w} \in B, && \text{/* Completeness */} \\ & p(B) \leq p(w_q) \text{ or } |B| = 1, && \text{/* Security */} \end{aligned}$$

and checks all the passwords in B using ExChk. We let $B(\tilde{w})$ denote the solution of the optimization problem induced by the checker OpChk on input string $\tilde{w}$.

Observe that in addition to completeness, the constraints in the optimization problem enforce the condition that $p(B(\tilde{w})) > p(w_q)$ only if $|B(\tilde{w})| = 1$. Thus,

OpChk ensures that the only balls with aggregate probability exceeding $p(w_q)$ are singletons (containing one high-probability password). The intuition here is that if we never allow a query to cover more probability mass than that of the q^{th} most popular password, then adversary $\mathcal{A}^*$ must select as its q queries the passwords $\{w_1, w_2, \ldots, w_q\}$. As these passwords define singleton balls, it follows that $\mathcal{A}^*$ will achieve exactly the same success probability as it would for an exact checker, and thus $\lambda_q^{\text{fuzzy}} = \lambda_q$.

We now give theorem statements showing that OpChk is indeed optimal in the sense that: (1) It achieves free corrections, meaning $\Delta_q = 0$ and, equivalently, $\lambda_q^{\text{fuzzy}} = \lambda_q$, for suitable q, and (2) Over all checkers with $\Delta_q = 0$, it achieves optimal utility, i.e., the highest possible probability of correcting a typo. The proofs of the following theorems appear in Appendix B.6.

Theorem 3.5.1 (Free Corrections Theorem) *Fix some password distribution p with support $\mathcal{P}$, a typo distribution τ, $0 < q < |\mathcal{P}|$ and an exact checker* ExChk. *Then for* OpChk *with any set of correctors $\mathcal{C}$, it holds that $\lambda_q^{\text{fuzzy}} = \lambda_q$.*

Theorem 3.5.2 (Optimality of OpChk) *Fix $q > 0$, a distribution pair (p, τ), and a corrector set $\mathcal{C}$. Define* OpChk *to work over $\mathcal{C}$ and let* Chk *work for a set of correctors $\mathcal{C}' \subseteq \mathcal{C}$. If $\Delta_q(\mathsf{Chk}) = 0$, then* $\mathsf{Utility}(\mathsf{Chk}) \leq \mathsf{Utility}(\mathsf{OpChk})$.

The free correction theorem applies with respect to an optimal attacker. We caution that it does not imply that for *any* attacker there is no security loss. Rather it is easy to give examples of password settings for which there exists some attack that achieves a speed-up due to tolerance. This attack, whatever it may be, cannot perform better than the optimal one. Finding good analogs to OpChk and the

free correction theorem for non-optimal attackers is an interesting, open research problem. We empirically investigate in the next section the relative performance of some non-optimal attacks, showing that these achieve no meaningful speed-up due to typo tolerance.

3.6 Practical Typo-Tolerant Checkers and their Security

In the previous section, we presented an optimal checker OpChk that achieves the maximum acceptance utility that is achievable with no loss in security (relative to optimal attacker). Unfortunately, OpChk is hard in general to implement, as it requires exact knowledge of the distribution pair (p, τ), which is not practically obtainable in most settings.

Here, we explore checkers that do not rely on exact distribution knowledge and are simple to implement. The first tries all corrections in some checker set. The latter two incorporate heuristics to try to avoid balls with high aggregate mass; these are directly inspired by the results regarding OpChk. As we show experimentally, our checkers can achieve high acceptance utility with minimal security degradation, and the heuristics help reduce security loss even against adversaries with exact knowledge of the probability distribution p. We also investigate the security of these checkers against more realistic adversaries that must themselves estimate the distribution p. For these adversaries our results here suggest that typo tolerance does not really help adversaries at all because of the difficulty of getting estimates right.

The tolerant checkers. For the following, let $\tilde{w}$ denote the input to the checker and $\hat{B}(\tilde{w})$ the ball of potential passwords to check as defined by the set of correctors

$\mathcal{C}$ for the checker. Presented in increasing order of sophistication (and similarity to OpChk), the checkers are:

- *Check-always construction* (Chk-All): This checker checks all passwords in $\hat{B}(\tilde{w})$. Among the three checkers presented here, it achieves the greatest acceptance utility—and, conversely, the largest potential security degradation.
- *Blacklist construction* (Chk-wBL). This checker uses a blacklist L of (ostensibly high-probability) passwords. It checks $\tilde{w}$ and every other password $w \in \hat{B}(\tilde{w})$ such that $w \notin L$. Blacklisting in Chk-wBL aims to prune or eliminate non-singleton balls with high aggregate probability (as OpChk does). In our experiments, we use for the blacklist the 1,000 most popular passwords in RockYou, although one could use other blacklists as well, such as Twitter's banned password list [102].[6]
- *Approximately optimal construction* (Chk-AOp). This checker heuristically approximates OpChk. It estimates the distribution p of passwords using the empirical distribution of the RockYou password leak, and the distribution τ of typos using the empirical distribution learned from our MTurk study (see Figure 3.2). We denote these empirically derived distributions respectively by $\tilde{p}$ and $\tilde{\tau}$. Chk-AOp computes $B(\tilde{w})$ using the constraints used by OpChk (see the last section), but under the empirical distribution pair $(\tilde{p}, \tilde{\tau})$, rather than the (generally unknown) true distribution pair (p, τ). We set $q = 10^3$ for our experiments with Chk-AOp. We note that for the correction set sizes we consider, $c \leq 5$, solving the optimization problem is fast, as only 2^c possibilities for $B(\tilde{w})$ must be considered.

We will investigate these checkers for typo correction sets $\mathcal{C}_{\text{top2}} = \{\text{swc-all}, \text{swc-first}\}$,

[6] We emphasize that the blacklist is only used for typo corrections: we do not assume users are restricted from registering blacklisted passwords.

$\mathcal{C}_{\text{top3}} = \mathcal{C}_{\text{top2}} \cup \{\mathsf{rm\text{-}last}\}$ and $\mathcal{C}_{\text{top5}} = \mathcal{C}_{\text{top3}} \cup \{\mathsf{rm\text{-}first}, \mathsf{n2s\text{-}last}\}$. In terms of utility, we know from the second Dropbox study (Section 3.4) the improvements obtained when using Chk-All with $\mathcal{C}_{\text{top3}}$. The other two constructions will obtain slightly less utility due to the fact that some corrections will not be checked.

Our preliminary analysis, however, suggests that this utility reduction will be slight: both strategies, by design, prevent corrections only to popular passwords, which are rarely induced by typos in the first place (see Section 4.6.1). For example, we can simulate acceptance utility for a given checker as defined in Section 3.5 by letting p be defined to be the RockYou empirical distribution and τ to be the empirical frequencies of typo types observed. Then for $\mathcal{C}_{\text{top3}}$ we have that the blacklist and approximately optimal strategies only reduce utility by 0.03 percentage points and 0.08 percentage points, respectively.[7]

Implementation considerations. The checkers above are all easy to implement, but care must be taken to optimize performance and ensure timing attacks do not arise. Generally, each checker should first run $\mathsf{ExChk}(\tilde{w})$ since this must always be computed. If that fails, then a constant-time check of the remainder of the ball should be performed. This involves running ExChk for the maximum number of checks that could occur for any $\tilde{w}$, i.e., $|\mathcal{C}|$. If implemented in this manner, timing and other side-channels will only potentially leak that a user made a typo, but nothing else about their password. Users that correctly input their passwords experience no performance degradation compared to existing systems.

If one instead does not use a constant time implementation, for example just

[7]The absolute acceptance utilities for $\mathcal{C}_{\text{top3}}$ in these simulations are 0.9628, 0.9625, and 0.9620. But the low overall rate of typos in the MTurk experiments means that exact checking here obtains 0.9564 acceptance utility already, which is significantly less than what is implied by our Dropbox measurements.

running a check for each string in $B(\tilde{w})$, then timing side channels will arise that leak partial information about a user's password. For example, checking a singleton ball (which is induced by some inputs and not other inputs for Chk-wBL and Chk-AOp) would be faster than checking a ball with multiple passwords. Thus the side-channel would reveal whether the user entered a high-probability password.

Security evaluation. In the remainder of this section we evaluate the security of our schemes against two types of attacker:

(A) **Exact-knowledge attackers:** We start by evaluating security of the constructions in the face of attackers that (unrealistically) know the precise distribution from which passwords are drawn. We will use a range of simulated password distributions and adversarial query budgets.

(B) **Estimating attackers:** We will then turn to more realistic attackers that do not have exact knowledge of the password distribution. Our evaluations will show that in this context an attacker attempting to take advantage of tolerant checking, even when they know the precise checker, can be quite error-prone: attackers can even do worse than naive approaches that just guess the most probable passwords in order.

Our approach for these analyses will be to utilize different password leaks to simulate true password selection. We will use the RockYou, phpBB, and Myspace leaks for these purposes. These leaks contain respectively the passwords of more than 32 million, 255,421, and 41,545 users of three different websites. Below when we say the RockYou, phpBB, or Myspace distribution we mean sampling according to the empirical distribution given by the indicated leak. Note that this means for some analyses we will use RockYou both within the designs of Chk-wBL and

Chk-AOp as well as to test those designs' security, optimistically modeling that a "best-case" estimate of the distribution is known to the checker. While we could use a holdout set (sampled from RockYou without replacement, for example) to be more realistic, we instead simply perform analyses using the independent Myspace and phpBB data sets and report all of them for completeness.

3.6.1 Security against exact-knowledge attackers

We now evaluate the security of our constructions against attackers that have exact knowledge of the password distribution. Thus in this section we assume that the adversary knows not only the exact functioning of the checker being used (i.e., what typos it corrects for any submitted password), but also the precise distribution of passwords. The latter is a conservative assumption. Attackers in practice will lack such knowledge and we are therefore measuring worst-case security from this point of view.

We will focus on the greedy success rate increase $\lambda_q^{\text{greedy}} - \lambda_q$ for various values of the query budget q. We will also report on λ_q to put loss in context. To compute these values, we use the RockYou, Myspace, and phpBB distributions as a stand-ins to simulate a challenge distribution p. Since the optimal attacker is assumed to know the distributions exactly, in the exact checking setting she will simply guess the most probable q passwords. Here λ_q is straightforwardly computable (just sum the probabilities of the top q passwords in the challenge distribution). In the typo-tolerant settings, the attacker will construct a sequence of queries that achieves $\lambda_q^{\text{greedy}}$ using the algorithm given in Appendix B.5.

We start by comparing security for attackers given $q = 1,000$ queries across the

Challenge Dist.	Set	q = 10 All	wBL	AOp	Ex	q = 100 All	wBL	AOp	Ex	q = 1000 All	wBL	AOp	Ex
RockYou	$\mathcal{C}_{\text{top2}}$	0.03	0.00	0.00		0.15	0.05	0.00		0.51	0.32	0.00	
	$\mathcal{C}_{\text{top3}}$	0.22	0.03	0.00	1.95	0.56	0.14	0.00	4.50	1.41	0.86	0.00	11.23
	$\mathcal{C}_{\text{top5}}$	0.25	0.06	0.00		0.63	0.18	0.00		1.57	0.87	0.00	
phpBB	$\mathcal{C}_{\text{top2}}$	0.03	0.00	0.00		0.12	0.02	0.00		0.38	0.19	0.15	
	$\mathcal{C}_{\text{top3}}$	0.19	0.02	0.00	2.75	0.28	0.04	0.01	5.50	1.01	0.60	0.42	12.71
	$\mathcal{C}_{\text{top5}}$	0.20	0.03	0.01		0.31	0.05	0.02		1.13	0.72	0.47	
Myspace	$\mathcal{C}_{\text{top2}}$	0.03	0.01	0.00		0.15	0.12	0.03		0.49	0.45	0.35	
	$\mathcal{C}_{\text{top3}}$	0.17	0.06	0.02	0.79	0.62	0.46	0.32	2.86	2.46	2.21	1.59	9.54
	$\mathcal{C}_{\text{top5}}$	0.27	0.15	0.04		0.87	0.68	0.52		3.00	2.66	1.94	

Figure 3.6: Percentage improvements in an exact-knowledge adversary's success ($\lambda_q^{\text{greedy}} - \lambda_q$) for each setting (corrector strategy and correction set) and each of the challenge distributions, for $q \in \{10, 100, 1000\}$.

various distributions, schemes, and corrector sets. We are here being conservative: a query budget of 1,000 is very generous to an attacker, as many websites will lock an account after tens of failed requests. Figure 3.6 reports the optimal success probability λ_q against an exact checker for each setting, as well as the improvements $\lambda_q^{\text{greedy}} - \lambda_q$ for each typo tolerant checker, correction set pair. All numbers are reported as percentages. The worst degradation occurs for correcting all top five errors in the Myspace setting, where the attacker's success probability increases by 3% (from 9.5% to 12.5%). To put this worst-case in perspective, consider the naive (and incorrect) assumption that seems to underlie the criticism of typo tolerance [77]: it suggests instead a fivefold increase in attacker success when correcting five errors and thus an increase to 47.5% in the Myspace setting.

Elsewhere the increase is much smaller. For example, with Rockyou, one can always correct all top five errors with increase only 1.6%: an attacker's probability of success goes from 11.2% to 12.8%, a small improvement. This means that the adversary's first 1,000 guesses against a typo-tolerant checker do not benefit much from high-probability balls.

Moving from $\mathcal{C}_{\text{top2}}$ to $\mathcal{C}_{\text{top3}}$ can result in a relatively big jump in security loss. The reason is that the rm-last typo corrector admits many higher-mass balls than only correcting the considered capitalization errors. For example, adding a character to many popular passwords results in another popular password: `password` and `password1`, `abc123` and `abc1234`. Fewer such pairs exist for capitalization errors since fewer users choose passwords with capital letters. Indeed in the worst case for $\mathcal{C}_{\text{top2}}$ we see a just a 0.5% improvement in adversarial success compared to the 2.42% worst-case jump for $\mathcal{C}_{\text{top3}}$. It is no coincidence, perhaps, that Facebook's policy seems to align with Chk-All for $\mathcal{C}_{\text{top2}}$. Our measurements are the first reported validation of this policy.

Even though security loss is low for Chk-All, one may want to do better. The blacklist and approximately optimal checkers help. When the challenge distribution is RockYou the approximately optimal checker Chk-AOp is, in this case, actually optimally secure by construction, hence it suffers no security loss at all. Also note that Chk-wBL may benefit unduly by knowing exactly the top 1,000 passwords from RockYou. Thus the more important analyses are when tested on independent distributions. Here we see some loss as one would expect given that the attacker in these cases has, after all, more information about the challenge distribution than the checker. But now the loss is small, and Chk-AOp reduces the security loss compared to Chk-All by 0.53% on average over the Myspace and phpBB settings. Chk-wBL also reduces loss compared to Chk-All by 0.27% on average over Myspace and phpBB, but never improves security more than Chk-AOp.

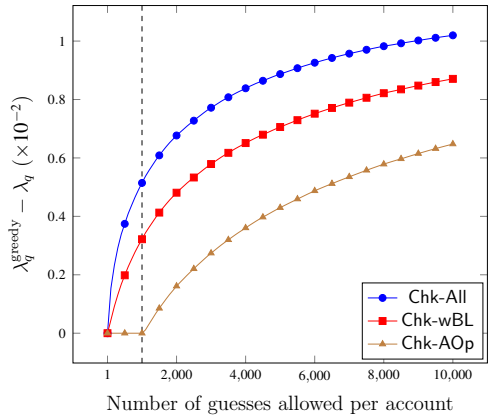
We now turn to what happens as q varies. In Figure 3.6 shows the attack success increases for the $q = 10$ and $q = 100$ cases. We note that the most realistic in practice is $q = 10$, since companies often will raise alarms after 10 consecutive

failed login attempts. Here we see that attackers benefit little from typo-tolerance, and our Chk-AOp reduces loss to 0.04% or less. Often it is zero.

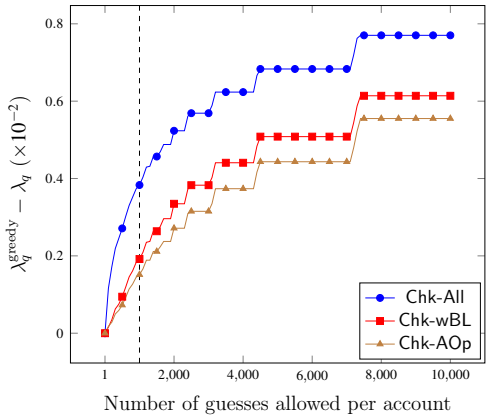
It is concievable that in some settings an attacker might be able to make more than $q = 1,000$ queries, which implies that our checker assumed too low of a bound on q. We focus for simplicity on $\mathcal{C}_{\text{top2}}$, a choice we expect many deployments to utilize, and show in Figure 3.7a the security loss using RockYou as the distribution for a range of $q \in \{1, 100, 200, 300, \ldots, 10{,}000\}$. We have drawn a vertical dotted line at $q = 1,000$, which was used by Chk-AOp as the expected query budget. As before, Chk-AOp has no loss below $q = 1,000$, and only after the attacker gets more than 1,000 queries does the attacker obtain an improvement over the exact checking case. Figure 3.7b shows the same type of chart but now for phpBB. This distribution leads to security loss seeing big discrete jumps for larger q, suggesting that at certain points the attacker can take advantage of new balls that just come in to play as higher mass than individual passwords. A chart for Myspace would exhibit similar trends as the one for phpBB, we omit it for the sake of brevity.

Observe that as q gets larger, the improvement $\lambda_q^{\text{greedy}} - \lambda_q$ flattens out in both charts. The attacker in the typo-tolerant cases runs out of heavily-weighted balls to take advantage of and ends up just querying passwords that cover only one (high-probability) guess. For RockYou, we see that the improvement is never more than 1% and, for phpBB, never more than 0.8%. This all suggests that even as q grows to values unlikely ever to arise in practice, typo-tolerance nevertheless does not improve the attacker's rate of success by much. We note that our blacklisting and approximately optimal checkers can be made even more conservative should one desire, by blacklisting more passwords or setting q larger, respectively.

As noted in the previous section, the greedy algorithm is known to provide



(a) The security loss as a function of q for challenge distribution RockYou and $\mathcal{C}_{\text{top2}}$.



(b) Difference in exact-knowledge adversary success against typo-tolerant schemes and exact checking, as a function of q for challenge distribution phpBB and $\mathcal{C}_{\text{top2}}$.

Figure 3.7: The security loss of typo-tolerant password checking with different values of q for perfect knowledge attacker (3.7a) and imperfect knowledge attacker (3.7b).

a good approximation for the weighted max coverage problem. Given that the password probabilities range over a very dense space, and our correction sets are quite small, we expect $\lambda_q^{\text{greedy}} \approx \lambda^{\text{fuzzy}}$. We do not have any theoretical proof for this claim, and leave analysis as an important question for future work. We can of course always bound the actual value of Δ_q via $\Delta_q \leq 1.582\,\Delta_q^{\text{greedy}} + 0.582\lambda_q$. So, for example with the RockYou challenge distribution, $q = 10$, and the $\mathcal{C}_{\text{top5}}$ corrector set we have that $\Delta_q \leq 0.0153$ as compared to $\lambda_q^{\text{greedy}} - \lambda_q = 0.0063$. We expect this three-fold decrease in relative security to be quite pessimistic: the better the greedy algorithm approximates the problem the worse the adjustment to compute Δ_q becomes.

	Attacker distribution	Challenge distribution		
		RockYou	phpBB	Myspace
ExChk	RockYou	11.23	3.21	9.34
	phpBB	8.10	12.71	1.81
	Myspace	3.57	3.32	9.54
Chk-All	RockYou	+0.51	+0.28	+0.25
	phpBB	+0.25	+0.38	+0.11
	Myspace	**-0.15**	**-0.02**	+0.49
Chk-wBL	RockYou	+0.32	+0.11	+0.20
	phpBB	+0.06	+0.19	+0.05
	Myspace	**-0.26**	**-0.20**	+0.46
Chk-AOp	RockYou	0.00	0.00	0.00
	phpBB	**-0.11**	+0.15	**-0.04**
	Myspace	**-0.27**	**-0.14**	+0.35

Figure 3.8: The top table shows the success rate of an attack against the exact checking scheme for the attacker-estimated distribution (row) used against the challenge distribution (column). The remaining tables show the *difference* between success rate of an attacker against the tolerant scheme and the exact checking scheme, for the indicated attacker-estimated and actual challenge distribution pairs. All values are in percentages.

3.6.2 Estimating attackers

We have so far considered attackers that have exact knowledge of the password distribution (even when the system designer may not). In practice such attackers do not exist, and instead adversaries must try to estimate the distribution of passwords. We refer to these as estimating attackers. As before, we assume adversaries know the exact checking algorithm in use.

We started by considering an adversary that estimates the password distribution using the Weir et al. probabilistic context-free grammar (PCFG) [39], a trained model of password distributions used to build effective crackers. However, our experiments with this showed that it provides poor efficacy in online guessing attacks, doing significantly worse than the approaches we describe below and, importantly, it did equally poorly against the typo-tolerant checkers in all settings.

We therefore turn to a different adversarial strategy for estimating the password

distribution. We measure the success rate of an attacker that uses one of the password leaks as its estimate of the distribution. This is a typical strategy in practice. We test these attacks against the other two distributions and for each of the exact checking, Chk-All, Chk-wBL, and Chk-AOp. The latter three use $\mathcal{C}_{\text{top2}}$. The security loss for all combinations are tabulated in Figure 3.8. (Note that the left-to-right diagonals reflect some of the results already shown for the exact-knowledge attacker in Figure 3.6.)

The improvement the attacker obtains when one switches to a tolerant checking system is never greater than 0.28%. More interestingly, in some cases the difference is negative, which means that the attacker did *worse* against the typo-tolerant scheme. This may be counterintuitive, but here the estimates the attacker makes about the distribution can often be wrong. This can lead her to choose a set of guesses that maximizes the total success probability according to her estimate but not according to the challenge distribution. We give an example for the curious reader in Appendix B.7.

In summary, our simulations here suggest that a carefully designed typo-tolerant checker will result in little to no security loss against realistic adversaries.

3.7 Conclusion

We presented the first treatment of typo-tolerant password authentication. We demonstrated, with large-scale, real-world experiments, that password typos are a real and common source of user errors in authentication systems. We found that a few types of typo-corrections account for an overwhelming number of password typos. We provided a formal framework for exploring typo-tolerant password

checkers, and focused on a class of them called relaxed checkers that are backwards-compatible with existing password hashing schemes. We showed, via what we call the free corrections theorem, that there exist relaxed checkers against which the best attack performs no better than the best attack against an exact checker. Unfortunately the construction requires exact knowledge of the password distribution. We therefore gave a number of practical typo-tolerant checkers inspired by it, and analyzed their security empirically, showing that one can easily obtain significant utility improvement with minimal or no security degradation.

In future work, we plan to investigate whether typo-tolerance will actually serve to improve overall security. Because allowing for password typos increases login success rates in benign scenarios, it may help to make adversarial login attempts stick out. This would strengthen the signals used to detect online password attacks as used in Internet-scale authentication systems.

CHAPTER 4

THE TYPTOP SYSTEM: PERSONALIZED TYPO-TOLERANT PASSWORD CHECKING

This chapter was published in ACM Conference on Computer and Communications Security (CCS) in 2017 [31].

4.1 Introduction

Passwords remain the predominant means of authenticating users on both computers and the web — however studies show that users persistently pick weak passwords [45, 68, 67]. This phenomena is often ascribed to users selecting easy-to-remember passwords; however a number of studies [75, 74, 72] highlight that strong passwords are also more difficult to type. To increase password usability, some companies [76, 77, 78] allow authentication under a small set of common typos. For example, Facebook permits capitalization errors in the first letter or accidental caps lock errors.

Motivated by this, Chatterjee et al. [30] recently initiated the academic investigation of typo-tolerant password checking. In a 24-hour study at Dropbox, they found that a small set of easy-to-correct typos accounted for over 9.3% of failed login attempts, and 3% of the total users turned away — underscoring the burden that typos represent to both users and the companies that ultimately lose out on user engagement due to them. To increase usability, the authors advocate an approach they call 'relaxed checking'. They establish a handful of the most frequently occurring typos across a user population, and build corrector functions to rectify those particular typos on behalf of the user at the time of authentication

(e.g., flipping the case of all letters to correct a caps lock error). The authors show empirically that for a carefully selected set of correctors, the resulting security degradation is minimal.

A limitation of this approach is that checking each correction requires applying a computationally intensive password hashing algorithm; as such the number of typos one may correct is inherently limited by performance constraints. While correcting the five most prevalent typos accounts for 20% of typos made by users [30], this still leaves the majority of password typos uncorrectable. Individual users may be totally neglected, should they frequently make a typo that is rare across the broader population of users. Users who choose complex, strong passwords are likely to fall into this neglected group.

We introduce a new approach to password checking: personalized typo-tolerance. In such a system, the password checking mechanism learns the typos commonly made by each user over time, storing them in a secure manner. After learning frequent typos, the system can check to see if a submitted password is either the one originally registered, or one of the learned variants. By tailoring typo-tolerance to the individual user, we aim to correct a larger set of typos than previously possible, while maintaining strong security guarantees.

Building a personalized typo-tolerant checking system requires care. The system should not begin accepting arbitrary incorrect passwords that are submitted — indeed this would enable potentially malicious users to register arbitrary passwords which allow access to an account. Therefore we need a policy dictating the types of errors that can be added to a cache of allowed typos, and a mechanism to enforce it. We must consider security in the face of remote guessing attacks as well as compromise of password databases, both being threats that frequently arise in

practice. This rules out simple schemes in which recent incorrect submissions are simply stored in the clear. Ideally, a scheme would be as secure as a conventional password-based checking system, meaning an attacker must perform as much work to compromise an account as they would have had there been no typo-tolerance.

We overcome these challenges and design a secure personalized typo-tolerant password checking system that we call TypTop. At a high level, the system works as follows. It maintains a set of allowed hashes, corresponding to the registered password and allowed typos of it. The user can successfully login by submitting either the password or an allowed typo. Initially, this set contains only the salted hashes of the registered password and some typos of that password that are considered likely across the population of all users. To personalize, TypTop adapts this set of typos over time by securely storing incorrect submissions encrypted under a public-key for which the associated secret key is, itself, encrypted under the registered password and previously allowed typos. Upon a subsequent successful login, the recent incorrect submissions can be decrypted and checked to see if they satisfy the policy regarding typos. If so, new salted hashes of the incorrect submissions, now considered as legitimate typos, can be added to the set of allowed hashes. To ensure the set does not grow too large, less frequently observed typos can be evicted. Future login attempts that make one of the frequent typos will be allowed, thereby avoiding the need for the user to retype their password.

Underlying TypTop is therefore a new kind of stateful password-based encryption scheme that ensures the plaintext state for a user can only be unlocked with knowledge of the registered password or one of the policy-checked cached variants of it. We introduce a formal security notion requiring that an attacker learns nothing about a sequence of password logins (including the number of logins, how

many variants were entered into the typo cache, or partial information about the passwords) given the state of the password system, unless the attacker can successfully mount a modified form of brute-force guessing attack in which it repeatedly hashes common passwords or typos of them and checks them against the hashes stored within the state.

We prove the security of our construction relative to this notion, and go on to analyze the brute-force guessing game to which we reduce security. We give criteria on password and typo distributions which, if met, mean that the attacker will gain no additional benefit by attempting to guess a stored typo. For such settings, we prove that the optimal strategy is a standard brute-force guessing attack against the registered password as if there were no additional password hashes of typos stored. We show empirically that real-world password and typo distributions meet the required criteria.

To gauge the potential efficacy of TypTop, we conduct a study using Amazon Mechanical Turk (MTurk) [91] in which we ask users to perform repeated logins using a password of their choosing.[1] In this way, we can analyze the types of errors made and the potential benefit of personalization compared to the prior relaxed checking approach with a fixed set of typo correctors. The analysis reveals that 45% of users would benefit from personalization, a 1.5x improvement over the 29% of users that benefit from the top 5 correctors from [30].

We implemented a prototype of TypTop for Unix systems including Linux and Mac OS using the pluggable authentication module (PAM) framework. The prototype enables typo-tolerance for all password-based authentications managed by the operating system. We report on the initial deployment with 25 users, and

[1] All our study designs were reviewed by our university's IRB.

while further studies will be needed to assess generalizability of our results, they so far indicate that TypTop significantly benefits users that often mistype their passwords in ways not covered by prior correction mechanisms. Our prototype is open-source and publicly available.[2]

4.2 Background and Related Work

In traditional password-based authentication schemes (PBAS), a user initially registers a user name and a password with the system. The password is stored in some protected form (typically a salted password hash). On subsequent login attempts, the user re-enters their password which is then compared to the stored password; authentication is granted only if these match. A formal definition of PBAS schemes is given in Section 5.2.

Password distributions and guessing attacks. Measurement studies [67, 10, 45] and password leaks such as [81] show that users frequently pick weak passwords, with a large number of users sharing a relatively small set of passwords at the head of the password distribution. This leaves them vulnerable to guessing attacks (see Section 5.2).

The reason users persistently pick such weak passwords is often cited as ease of memorability. However studies by Keith et al. [74, 75] indicate that the rate at which typos are made approximately doubles for complex passwords; similarly Mazurek et al. [82] show via a large-scale study that login errors are correlated with stronger passwords, and suggest that users pick weaker passwords due to their ease of typing. Work by Shay et al. [71] finds a similar correlation between length and

[2] https://typtop.info/

rate of typo occurrence for CorrectHorseBatteryStaple-type passphrases [85]. For a more detailed discussion of these related works, see Chapter 3.

Typo-tolerant password checking. Several web services [76, 77, 78], allow (or used to allow) small typographical errors in their passwords. In the Chapter 3, we provide the first formal treatment of typo-tolerant password checking for user-selected passwords. To recall, with an experiment conducted on MTurk, we show that 20% of typos can be corrected by a small set of corrector functions (e.g., applying caps lock or switching the case of the first letter). We give *relaxed checking*, which allows authentication under a small number of easily correctable typos. Now when a password fails its initial authentication check, a set of (e.g., 5) corrector functions representing common typos are applied to the entered string; the user is allowed to authenticate if any of these corrections matches the registered password. Relaxed checking with a carefully chosen set of corrector functions can achieve a significant improvement in utility with minimal degradation in security.

While relaxed checking allows for the secure correction of 20% of typos, this still leaves the majority of typos uncorrected. Furthermore, the corrector functions utilized are based on common typo behavior across the population of users, as opposed to that of the individual. In this chapter, we explore a new approach — personalized typo-tolerance — which allows us to correct a greater proportion of typos by tailoring typo-tolerance to the individual user.

4.3 Personalized Typo Tolerance

We introduce *personalized* typo-tolerant password checking which adapts, over time, to correct the specific typos made by an individual user. We begin by defining

the abstract components of such a scheme, as well as the utility and the security goals.

Passwords and typos. We let $\mathcal{S}$ denote the set of strings which users may choose as passwords (e.g., the set of ASCII strings up to some maximum length, say ℓ). We let p be a distribution which models user selection of passwords, so $p(w)$ denotes the probability that w is the password chosen by a user, and let $\mathcal{P}$ denote the support of p, which represents the set of user passwords. We let $w_1, w_2, \ldots$ denote the passwords in $\mathcal{P}$ ordered in descending order of probability.

A typo-tolerant PBAS which allows authentication under *any* string is clearly insecure. As such we need a means to distinguish legitimate user typos from unrelated strings. We first convert both the password w and possible typo $\tilde{w}$ into their key press representation [30], and then compute the Damerau-Levenshtein distance [86, 87] of these representations, which we denote $\mathrm{DL}(w, \tilde{w})$. We view $\tilde{w}$ as a legitimate typo of w if this distance is at most some small fixed parameter δ; here we set $\delta = 2$. We let $\{\tau_w\}$ be a family of distributions over typos for each password w, so $\tau_w(\tilde{w})$ is the probability that the password $w \in \mathcal{P}$ is typed as $\tilde{w} \in \mathcal{S}$. Note that $\tau_w(w)$ denotes the probability that w is typed correctly. Together these components (p, τ) define an *error setting*.

Looking ahead, we will conservatively assume that an attacker has precise knowledge of the underlying error setting when we analyze the security of TypTop with respect to guessing attacks. In practice, TypTop uses a state of the art password strength estimator, zxcvbn [103], to estimate password guessability.

Adaptive checkers. An adaptive password checker Π is a stateful PBAS — that is to say $\Pi = (\mathsf{Reg}, \mathsf{Chk})$ is a pair of algorithm defined as follows:

- Reg is a randomized algorithm which takes as input a password w, and outputs an initial state s_0 for Π.
- Chk is an algorithm (possibly randomized) which takes as input a string $\tilde{w}$ and a state s, and outputs a bit b and an updated state s'. An output $b = 1$ means authentication is granted.

Our definition is analogous to the formalization of standard (non-adaptive) PBAS given in Chapter 3; their definition may be recovered by keeping the state constant across invocations of Chk. We call a non-adaptive PBAS an *exact checker* if it outputs $b = 1$ only if the correct password is entered exactly.

We require that a password checker is *complete*, which is to say that the probability that a user successfully authenticates when he enters his correct password is one. Additionally we desire our typo-tolerant checker to authenticate under as many typos made by a legitimate user as possible (subject to security constraints). We will measure the utility of a typo-tolerant PBAS in terms of the fraction of typos accepted by the password checker across all users. By this measure, the utility of an exact checker is always 0.

Guessing attacks. Guessing attacks against PBAS schemes come in two key flavors: online and offline attacks. In the former, an attacker uses the login API of the system to submit different password guesses in an attempt to impersonate a user. The attacker might target a specific user in what is known as a vertical attack, or may try common passwords against the accounts of many users (known as a horizontal attack). Various countermeasures can be deployed to mitigate these attacks, including locking the account after a number of (e.g., 10) incorrect password submissions, and slowing down responses for login attempts that ap-

pear unsafe based on contextual information (for example, those originating from suspicious IP addresses).

In offline attacks we assume that the attacker has compromised the database of stored PBAS states, and attempts to brute-force guess the underlying values. In contrast to online attacks, the number of guesses made by an offline attacker is limited only by the computational power they are willing to expend. We analyze the security of TypTop in the face of both forms of attack in Section 4.5.

4.4 The TypTop Design

We first give an overview of TypTop, and then detail its components more fully. TypTop uses a *typo cache* and an encrypted *wait list*. The typo cache securely stores the set of strings under which the user is allowed to authenticate; namely their registered password plus a number of previously accepted typos of that password. The wait list is a public-key encryption of recent incorrect password submissions that were not the registered password or one of the typos already in the typo cache. The secret decryption key for the wait list is, in turn, encrypted using the registered password and (separately) under each of the cached typos. When a login attempt's password submission is accepted — either it was the registered password or one of the previously accepted typos — the wait list can be decrypted and processed according to some typo cache policy. The latter defines which incorrect submissions should be allowed into the typo cache. A diagrammatic view of TypTop's Chk procedure is given in Figure 4.1.

Underlying components. We begin by defining the primitives utilized by TypTop. A public-key encryption (PKE) scheme $\mathsf{PKE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ is a triple

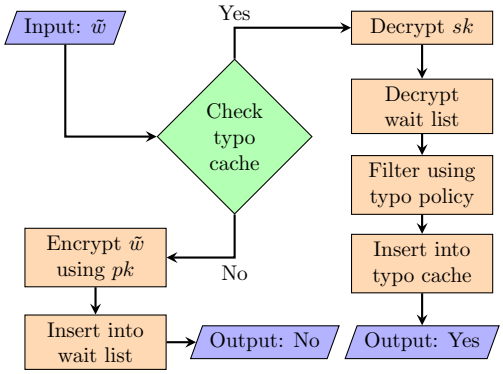


Figure 4.1: Diagram showing TypTop's approach to personalized typo-tolerant password checking.

of algorithms. The key generation algorithm $\mathcal{K}$ takes random coins as input and outputs a public / secret key pair $(pk, sk) \leftarrow\!\!\$\, \mathcal{K}$. The randomized encryption algorithm $\mathcal{E}$ takes as input a public key pk and a message $m \in \mathcal{M}_{\mathcal{E}}$ (where $\mathcal{M}_{\mathcal{E}}$ denotes the message space), and outputs a ciphertext $c \leftarrow\!\!\$\, \mathcal{E}_{pk}(m)$. We let $\mathcal{C}_{\mathcal{E}}$ denote the ciphertext space. The deterministic decryption algorithm $\mathcal{D}$ takes as input a secret key sk and a ciphertext c and outputs a message $\tilde{m} \in \mathcal{M}_{\mathcal{E}} \cup \{\perp\}$. We use a PKE scheme with perfect correctness, meaning that the probability an honestly generated ciphertext decrypts to the correct message is one.

A password-based encryption (PBE) scheme $\mathsf{PBE} = (\mathsf{E}, \mathsf{D})$ is a pair of algorithms defined as follows. The randomized encryption algorithm E takes as input a password $w \in \mathcal{P}$ (the set of all allowed passwords) and a message $m \in \mathcal{M}_{\mathsf{E}}$ (the message space), and outputs a ciphertext $c \leftarrow\!\!\$\, \mathsf{E}_w(m)$, where we let $\mathcal{C}_{\mathsf{E}}$ denote the ciphertext space. The deterministic decryption algorithm D takes as input a password w and a ciphertext c and outputs a message $\tilde{m} \in \mathcal{M}_{\mathsf{E}} \cup \{\perp\}$. We require PBE to be perfectly correct. A conventional symmetric encryption scheme is the same as a PBE scheme, except that it assumes uniform bit strings of some length κ as keys.

We let $\mathsf{PBE}[\mathsf{SH},\mathsf{SE}] = (\overline{\mathsf{E}},\overline{\mathsf{D}})$ denote the canonical PBE scheme that works as follows. The scheme utilizes a random oracle with signature $\mathsf{SH} : \{0,1\}^* \to \{0,1\}^\kappa$ and a conventional symmetric encryption scheme $\mathsf{SE} = (\mathsf{E},\mathsf{D})$ that uses keys of length κ bits. Then $\overline{\mathsf{E}}(w,m)$ first chooses a fresh salt $\mathsf{sa} \leftarrow\!\!\$\ \{0,1\}^{\ell_{\text{salt}}}$ for some suitably large ℓ_{salt}, computes $c = \mathsf{E}(\mathsf{SH}(\mathsf{sa}\|w), m)$, and outputs (sa, c). Decryption works in the obvious way.

Later, we will assume the message space associated to a PKE scheme unambiguously supports passwords of length up to some parameter ℓ (passwords will be unambiguously padded to this max length), a distinguished empty string symbol ε, and the state space of the caching scheme. We assume that encryptions of passwords and the empty string are of equal length. We assume PBE has a message space containing a typical representation of the PKE scheme's secret keys.

Finally, we let $\mathsf{Perm}(t)$ denotes the set of all permutations on $\mathbb{Z}_t$. Looking ahead, we shall set t to be the number of typos stored in the typo cache, and regularly apply a random permutation to randomize the order of elements in the cache. For our scheme, t will be small, making random permutations on $\mathbb{Z}_t$ easily sampleable and representable.

The details. A pseudocode description of TypTop's adaptive typo-tolerant password checking scheme $\Pi = (\mathsf{Reg},\mathsf{Chk})$ appears in Figure 4.2. TypTop uses a caching scheme to determine which entries in the wait list are integrated into the cache. In the figure we detail a probabilistic least frequently used (PLFU) caching scheme, but TypTop works modularly with other caching schemes as discussed later in this section.

The state of the adaptive checking scheme s consists of a public key pk, the

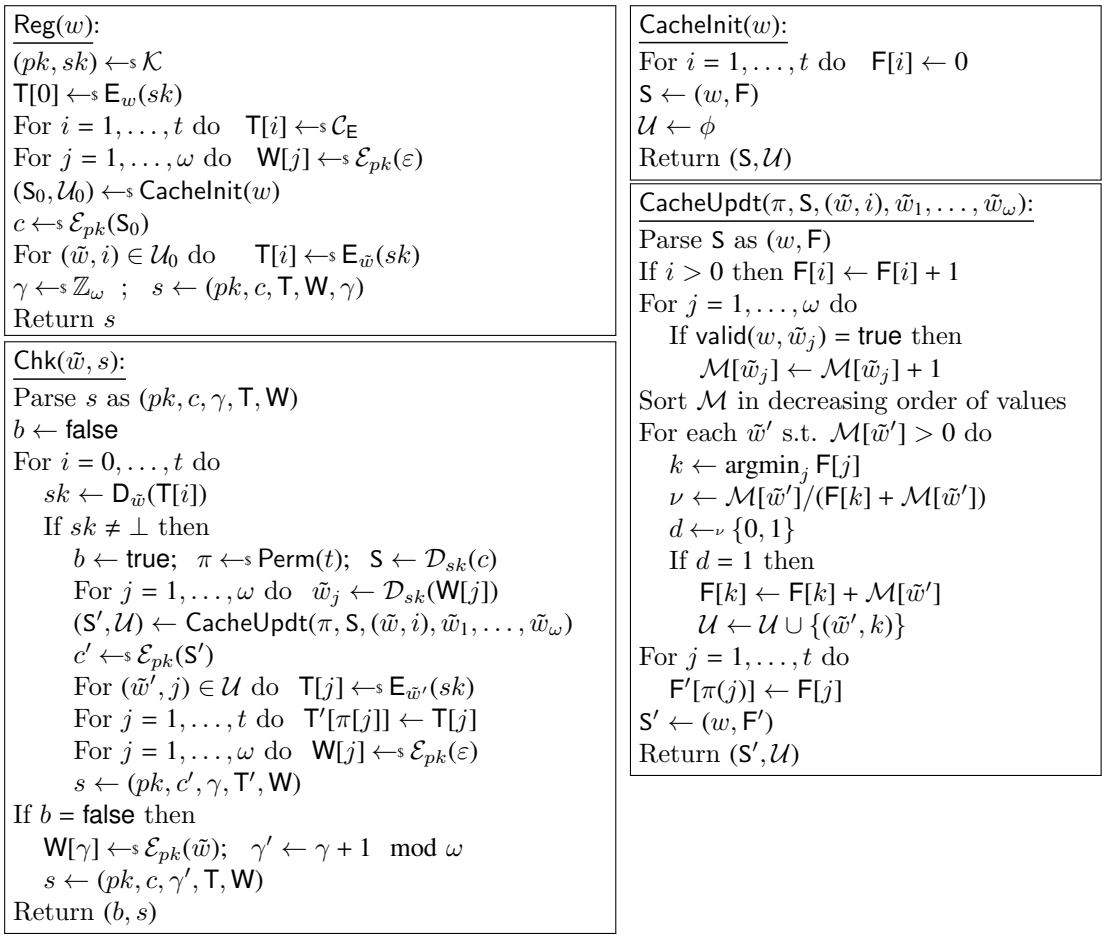


Figure 4.2: Our adaptive password checking scheme $\Pi = (\mathsf{Reg}, \mathsf{Chk})$ using a modified least-frequently used caching policy. The latter uses a function valid that checks whether a string should be considered for entry into the typo cache (e.g., checking whether a string lies within an edit distance threshold of the true password).

encryption of the caching scheme's (plaintext) state S, a typo cache T, an encrypted wait list W, and an index γ which is a pointer to the next wait list entry that should be used. The typo cache consists of up to t PBE encryptions of the secret key sk corresponding to pk, where t is a parameter of the scheme. The typo cache is initially filled with random ciphertexts, unless otherwise indicated by the caching scheme in use (e.g., one may want to warm up the cache with possible typos as discussed below). The wait list consists of up to ω PKE encryptions of incorrect password submissions. For the wait list we use a simple least-recently entered

eviction policy, accomplished by having index γ wrap around. To force a wrap around would require ω incorrect submissions before a correct one, so in practice we can set ω to be equal to a lockout threshold (such as 10). The wait list is initialized with encryptions of the empty string symbol ε, and cleared in the same manner. The index γ is initialized to a random value in $\mathbb{Z}_\omega$.

After every change to the typo cache, a random permutation $\pi \leftarrow\!\!\$\ \mathsf{Perm}(t)$ is used to permute the order of the cached typos. This is to ensure that even if an adversary knows the typos likely to be made by a user, he will not know at which position each typo lies in the cache. This will have ramifications for offline security, making the guessing game harder for particular distributions. See Section 4.5.

Caching schemes. TypTop maintains a set of cached typos which evolves over time based on the users' login attempts and the caching policy in use. We require that the set of cached typos are distinct, and that the real password is never cached as a typo; this maximizes the number of typos we can tolerate for a given cache size. We abstract out the process of initializing and updating the typo cache via a stateful caching scheme $\mathsf{Cache} = (\mathsf{CacheInit}, \mathsf{CacheUpdt})$ defined as follows. The algorithm $\mathsf{CacheInit}$ takes as input a password w, and outputs an initial state for the caching scheme S_0 and a set $\mathcal{U}_0$ of typo / index pairs $(\tilde{w}, i)$, indicating that the typo $\tilde{w}$ should be stored at the i^{th} position in the initial cache. The algorithm $\mathsf{CacheUpdt}$ takes as input the caching scheme state S and a list $(\tilde{w}_1, \ldots, \tilde{w}_\omega)$ of candidate replacement typos (in our case, drawn from the wait list) plus any other information required to implement the caching policy (e.g., their frequencies). It outputs an updated state S' and a set $\mathcal{U}$ indicating the replacements to be made.

The checker Π is designed so that any caching scheme of choice may be dropped in. The set of caching schemes we consider is given in Figure 4.3. The simplest

Scheme	Replacement Policy
LRU	Replace least recently used typo with $\tilde{w}_n$
LFU	Replace least frequently used typo and associated frequency with $(\tilde{w}_n, f_{\tilde{w}_n})$
PLFU	Replace least frequently used typo $\tilde{w}_o$ and associated frequency with $(\tilde{w}_n, f_{\tilde{w}_n} + f_{\tilde{w}_o})$ with probability $\frac{f_{\tilde{w}_n}}{f_{\tilde{w}_n} + f_{\tilde{w}_o}}$
MFU	Make necessary replacements to ensure t most frequently used typos lie in cache
Best-t	Initialize cache with t most probable typos based on typo model; never replace

Figure 4.3: Table summarizing the caching schemes considered. Here $\tilde{w}_n$ denotes the wait list typo being considered for inclusion in the cache, f denotes the frequency count of the typo in subscript, and t denotes the cache size.

is a least recently used (LRU) caching scheme $\mathsf{CacheLRU} = (\mathsf{InitLRU}, \mathsf{UpdateLRU})$ which maintains a list of typo cache indices ordered by how recently they were entered by the user; when we update the cache, the last (and least recently used) entry in the cache is evicted and replaced with the most frequently observed entry in the wait list. LRU ignores the frequency with which a user authenticates under a cached typo.

We consider three other strategies that take this frequency into consideration. The simplest scheme is the LFU policy, which performs cache updates by replacing the least frequently used cache typo with the most frequently observed wait list entry. The newly added typo has its frequency set to the number of times it appeared in the wait list.

A potential drawback of this approach is that we replace a cached typo each time we update, and so may inadvertently replace a typo that the user makes reasonably often (and is thus good to keep in the cache) with an anomalous typo from the wait list which they are unlikely to use again. We therefore give a probabilistic-LFU (PLFU) scheme, which performs cache updates as follows. First the frequencies of the least frequently used typo in the cache $\tilde{w}_o$ and the most frequently observed typo in the wait list $\tilde{w}_n$ are compared, and we replace the former with

the latter with probability $\nu = f_{\tilde{w}_n}/(f_{\tilde{w}_n} + f_{\tilde{w}_o})$, where f denotes the frequency count of the typo in subscript in the wait list (for $\tilde{w}_n$) or the typo cache (for $\tilde{w}_o$). If such an update occurs, the frequency of the newly cached typo is set to $f_{\tilde{w}_n} + f_{\tilde{w}_o}$. This process is repeated for each of the unique typos in the wait list in descending order of their frequency. This both serves to decrease the probability that a useful cached typo is replaced unnecessarily, and increases the likelihood that typos which are observed repeatedly in low frequencies over different login attempts are cached; we give a detailed discussion in Appendix C.1.

The above schemes require $|\mathsf{S}| \in \mathcal{O}(()\, t)$ space for caching; desirable in our construction since storing more data in the state of Π could negatively impact efficiency. In settings where this is less of a concern (e.g., authentication to personal devices) and we can afford to maintain a larger state, we can employ a most frequently used (MFU) caching policy which records the frequency of *all* valid typos made by a user so far. The t most frequent typos are maintained in the cache.

As a benchmark against which to compare the utility benefit of adaptive checking, we also consider a static caching policy Best-t: fill the cache of a given password w with its t most likely typos according to some typo model, and then never update the cache. Looking ahead, we will build a typo model from measurements of typos made by users.

Admissible typos. As discussed in Section 5.2, care must be taken when deciding which typos are cached. As such we use a procedure valid to test whether an entry in the wait list should be input to $\mathsf{CacheUpdt}$ as a candidate for inclusion. Our policy applies three restrictions. Firstly, we set a threshold d, and only consider a typo $\tilde{w}$ of a password w for inclusion if $\mathrm{DL}(w, \tilde{w}) \leq d$. For TypTop we use

$d = 1$ unless stated otherwise, allowing us to capture the caps lock errors, single substitutions, deletions and transpositions which studies indicate account for 46% of typos made by users [30].

Secondly, we wish to avoid including easily guessable typos in the cache which may speed up guessing attacks — for example `Password1#` may be mistyped as `Password1`, but the latter requires only 8 attempts to guess as opposed to nearer 1,000 for the former (as estimated by zxcvbn [103]). As such we impose two password strength checks with associated thresholds m, σ. For a typo $\tilde{w}$ of a password w to be considered admissible, $\tilde{w}$ must be such that both $\mu_{\tilde{w}} \geq m$ and $\mu_{\tilde{w}} \geq \mu_w - \sigma$, where μ denotes the strength estimate of the password / typo in subscript. The first condition ensures that the most easily guessed typos are never cached, while the second prevents the caching of typos significantly more guessable than the real password. We will use $m = 10$ and $\sigma = 3$ unless otherwise specified; a justification for these parameter choices is given in Section 4.6.1. To estimate guessability, we use zxcvbn due to its accuracy and ease of deployment; one could also use other strength estimators such as those based on neural networks [104].

Warming up caches. For all of the adaptive caching schemes described above, the user must make a typo at least once before it is considered for inclusion in the cache — for example, when the typo cache is "cold" immediately after registration, no typos will be tolerated. As such we consider initializing the typo cache T with probable typos of the registered password; a process we call "warming up" the cache. We build an empirical typo distribution using data collected via an MTurk study (detailed in Section 4.6.1), and the data released with [30]. We then fill the cache of a given password with its t most likely typos as indicated by the typo distribution, with their frequency counts set to 0. In contrast to relaxed

checking, these cached typos are chosen on a per password basis (as opposed to using population-wide corrector functions). We will always warm up the cache unless otherwise specified.

4.5 Security of TypTop

In this section, we analyze the security of TypTop within the two main threat models for password checking schemes described in Section 5.2: offline and online attacks.

In the offline setting an attacker gains access to the state of the checker, so we first analyze TypTop from a cryptographic viewpoint, showing that the state does not leak additional information about the user's password and login history. We give a formal security notion that captures this requirement, and provide a reduction showing that an attacker obtaining access to the state of TypTop learns nothing about the user's login behavior (including partial information about the password) unless they are able to brute-force guess the password or one of the typos active in the cache. With this in place, we consider the success probability of an attacker in such a brute-force attack. We show that for certain classes of error settings, the maximum advantage of an attacker against TypTop is no greater than that of an optimal attacker against the exact checker (who must brute-force guess the exact password in order to succeed).

We then analyze the online setting, which will be similar to the security analysis of the relaxed checking approach introduced by Chatterjee et al. [30]. The results indicate that security loss in the face of online attacks is minimal, with less security degradation than the prior approach [30].

In both online and offline settings, our analyses are with respect to an attacker who we conservatively assume has precise knowledge of the password distribution. In practice, where precise knowledge is unlikely, security will be even better than our analyses predict.

4.5.1 Cryptographic Security

Our security notion formalizes the following intuition. Consider an adversary that can obtain the state of a password checking system after registration plus some sequence of login attempts by a user. Then the adversary should not be able to distinguish this real state from one drawn at random from the state space of the checker $\mathcal{S}$, *unless* they are able to brute-force guess one of the passwords or typos allowed by the checking system at the point of compromise.

To this end we introduce some additional notation that will make defining security and our subsequent analysis simpler. For a given error setting (p, τ), we define an associated login transcript generator $\mathcal{T}$ to model a user's sequence of login attempts. Formally a login transcript generator $\mathcal{T}$ is defined to be a randomized algorithm that takes no input and outputs a sequence of passwords and typos which represent a user's selection of their password (the first password in the sequence) and subsequent login attempts, all sampled according to the appropriate distributions. A transcript checker associated to an adaptive password checker $\Pi = (\mathsf{Reg}, \mathsf{Chk})$ (see Section 5.2) is an algorithm $\mathsf{Checker}[\Pi]$ that takes as input a sequence of passwords and outputs a state value. The canonical such transcript checker, on input $w_0, \tilde{w}_1, \ldots, \tilde{w}_n$, runs $s_0 \leftarrow\!\!\$\ \mathsf{Reg}(w_0)$, and then computes $s_i \leftarrow\!\!\$\ \mathsf{Chk}(\tilde{w}_i, s_{i-1})$ iteratively for $1 \leq i \leq n$. It then outputs the final state s_n.

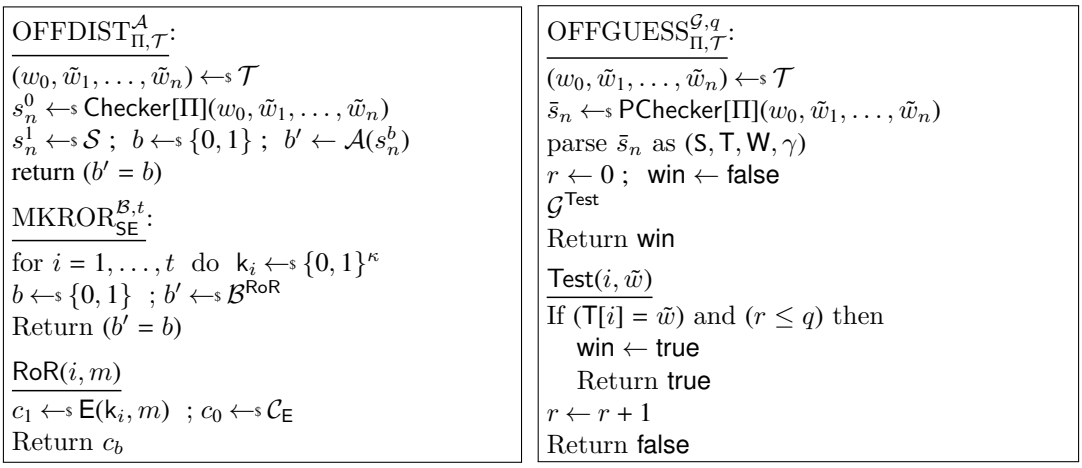


Figure 4.4: Cryptographic security games for adaptive password checking schemes, offline guessing attacks, and multi-key real-or-random symmetric encryption security.

Let $\Pi = (\mathsf{Reg}, \mathsf{Chk})$ be an adaptive password checker and let $\mathcal{T}$ be a transcript generator. Consider the game OFFDIST of Figure 4.4. We define the offline distinguishing advantage of an adversary $\mathcal{A}$ against $\Pi, \mathcal{T}$ to be

$$\mathsf{Adv}^{\mathrm{offdist}}_{\Pi,\mathcal{T}}(\mathcal{A}) = 2 \cdot \left| \Pr\left[\mathrm{OFFDIST}^{\mathcal{A}}_{\Pi,\mathcal{T}} \Rightarrow \mathsf{true} \right] - \frac{1}{2} \right|$$

where the probability is over the random coins used in executing the game. We will not provide strict definitions of security (e.g. using asymptotics), but rather measure concretely the advantage of adversaries given certain running times and query budgets.

The security model proposed here coincides with a one-time compromise of the system. A stronger model would perhaps allow adaptive compromises, observing multiple instances of the password checking state over time. We conjecture that Π meets such a definition but leave the analysis to future work.

Preliminaries. Before our analysis, we fix a number of further definitions that will be needed in the proofs.

We implement TypTop with the canonical PBE scheme

$\mathsf{PBE}[\mathsf{SH},\mathsf{SE}] = (\overline{\mathsf{E}},\overline{\mathsf{D}})$ as described in Section 4.4 which utilizes a symmetric encryption (SE) scheme SE and random oracle SH; adversaries in security games against this implementation of TypTop are given access to the random oracle accordingly.

We now define two security notions which we will require for the underlying SE scheme. The first is a multi-key real-vs-random security notion for symmetric encryption under chosen plaintext attack. Let $\mathsf{SE} = (\mathsf{E},\mathsf{D})$ be a SE scheme with associated ciphertext space $\mathcal{C}_{\mathsf{E}}$. The pseudocode description of the security game $\mathrm{MKROR}^{\mathcal{B},t}_{\mathsf{SE}}$ appears in Figure 4.4. This game tasks the attacker with determining whether it is receiving encryptions of a (chosen) message, or a random ciphertext, in a multi-key setting. We define the distinguishing advantage of an adversary $\mathcal{B}$ as

$$\mathsf{Adv}^{\mathrm{mkror}}_{\mathsf{SE}}(\mathcal{B},t) = 2 \cdot \left| \Pr\left[\, \mathrm{MKROR}^{\mathcal{B},t}_{\mathsf{SE}} \Rightarrow \mathsf{true} \,\right] - \frac{1}{2} \right| .$$

The security game for the familiar single-key real-vs-random security notion, which we denote $\mathrm{SKROR}^{\mathcal{B}}_{\mathsf{SE}}$, is obtained by setting $t = 1$ in the above definition. A straightforward hybrid argument shows that for any symmetric encryption scheme SE and adversary $\mathcal{B}$ in game $\mathrm{MKROR}^{\mathcal{B},t}_{\mathsf{SE}}$ running in time T, there exists an adversary $\mathcal{B}'$ in game $\mathrm{SKROR}^{\mathcal{B}}_{\mathsf{SE}}$ running in time $T' \approx T$ such that $\mathsf{Adv}^{\mathrm{mkror}}_{\mathsf{SE}}(\mathcal{B},t) \leq t \cdot \mathsf{Adv}^{\mathrm{skror}}_{\mathsf{SE}}(\mathcal{B}')$. By $T' \approx T$ (which we will also use in theorem statements below), we mean that the running time of $\mathcal{B}'$ is the same as that of $\mathcal{B}$ plus some qualitatively inconsequential overheads that can be derived from the proof.

We need one additional security property from our SE scheme: that of robustness, which ensures that no computationally efficient adversary can find two keys that both decrypt the same ciphertext. The notion of robustness for PKE

schemes was introduced by Abdalla et al. [105], and later extended in [106, 107]. Security notions and constructions for robust SE schemes are given by Farshim et al. [108]. We use a variant of their full robustness notion that is strictly weaker than full robustness, but which suffices for our purposes. Formally, let $\mathrm{ROB}^{\mathcal{R}}_{\mathsf{SE}}$ be the game that works as follows. The adversary $\mathcal{R}$ runs with no inputs and outputs $(\mathsf{k}, \mathsf{k}', m)$, i.e., a pair of keys and a message. The game then computes $c \leftarrow\!\!\$\ \mathsf{E}(\mathsf{k}, m)$ and $m' \leftarrow \mathsf{D}(\mathsf{k}', c)$. The game outputs true if $\mathsf{k} \neq \mathsf{k}'$ and $m' \neq \perp$. We define the advantage of an adversary $\mathcal{R}$ as $\mathsf{Adv}^{\mathrm{rob}}_{\mathsf{SE}}(\mathcal{R}) = \mathsf{Pr}\left[\, \mathrm{ROB}^{\mathcal{R}}_{\mathsf{SE}} \Rightarrow \mathsf{true} \,\right]$.

Finally we require a more standard real-vs-random ciphertext notion of security for a PKE scheme $\mathsf{PKE} = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ with associated ciphertext space $\mathcal{C}_{\mathcal{E}}$. The game $\mathrm{ROR}^{\mathcal{C}}_{\mathsf{PKE}}$ (not shown) first generates a key pair $(pk, sk) \leftarrow\!\!\$\ \mathcal{K}$ and a random bit b. It then runs adversary $\mathcal{C}(pk)$, who is given access to an oracle RoR to which it may query messages m. The oracle computes $c_1 \leftarrow\!\!\$\ \mathcal{E}(pk, m)$ and $c_0 \leftarrow\!\!\$\ \mathcal{C}_{\mathcal{E}}$ and returns c_b. Finally $\mathcal{C}$ outputs a bit b', and the game returns $(b = b')$. We define the distinguishing advantage of an adversary $\mathcal{C}$ as

$$\mathsf{Adv}^{\mathrm{ror}}_{\mathsf{PKE}}(\mathcal{C}) = 2 \cdot \left| \mathsf{Pr}\left[\, \mathrm{ROR}^{\mathcal{C}}_{\mathsf{PKE}} \Rightarrow \mathsf{true} \,\right] - \frac{1}{2} \right| .$$

Offline guessing attacks. With this in place, we now define the eventual target of our reduction: a guessing game in which the adversary obtains an oracle to make guesses against the password and its t cached typos. Observe that for the canonical transcript checker $\mathsf{Checker}[\Pi]$, the eventual entries in the typo cache (and wait list) associated to s_n depend not only on the input transcript $w_0, \tilde{w}_1, \ldots, \tilde{w}_n$ but also on whether a ciphertext in the typo cache is erroneously decrypted to something besides $\perp$ when using the *wrong* password. We can, however, rule out such an event using the robustness of the SE scheme (see definition above).

In order to simplify the subsequent analysis, we define a modified transcript checker $\mathsf{PChecker}[\Pi]$ that evolves the state of Π using only *plaintext* values. Crucially, rather than relying on a successful decryption to determine whether a password / typo lies in the cache, we may now simply compare the input to the plaintext cache values, thereby eliminating the negligible probability of erroneous state updates. The pseudocode for $\mathsf{PChecker}[\Pi]$ is given in Figure 4.5.

The game $\text{OFFGUESS}^{\mathcal{G},q}_{\Pi,\mathcal{T}}$ is given in Figure 4.4. The guessing advantage of an adversary $\mathcal{G}$ who makes at most q queries to the Test oracle is defined as $\mathsf{Adv}^{\text{offguess}}_{\Pi,\mathcal{T}}(\mathcal{G}, q) = \mathsf{Pr}[\text{OFFGUESS}^{\mathcal{G},q}_{\Pi,\mathcal{T}} \Rightarrow \mathsf{true}]$. The transcript generator and plaintext checker are first used to sample a set of cache values; the adversary succeeds if he can guess any password or typo which lies in the cache. Note that this game requires the adversary to specify which password each guess should be checked against. We measure the complexity of guessing adversaries in terms of the number of Test queries they make.

We assume without loss of generality that all adversaries make legitimate queries in their respective games (i.e., with values inside the appropriate domains, and with an index in the range $[0, t]$ for MKROR and OFFGUESS).

The analysis. Let $\Pi = (\mathsf{Reg}, \mathsf{Chk})$ denote TypTop's password checker, and fix some transcript generator $\mathcal{T}$. Our analysis will show that the $\text{OFFDIST}_{\Pi,\mathcal{T}}$ security of Π reduces to the guessing game $\text{OFFGUESS}_{\Pi,\mathcal{T}}$. We note that this analysis is independent of the specific caching scheme used by TypTop; the effect on security of different such schemes will be surfaced in Section 4.5.2 when we bound the success probability of an attacker in game $\text{OFFGUESS}_{\Pi,\mathcal{T}}$.

As the first step in our reduction, we introduce $\mathsf{PChecker}[\Pi]$ into game

$\text{OFFDIST}_{\Pi,\mathcal{T}}$, via an intermediate game $\overline{\text{OFFDIST}}_{\Pi,\mathcal{T}}$ defined to be identical except we replace $\mathsf{Checker}[\Pi]$ with $\mathsf{PChecker}[\Pi]$, and then use the values in the final plaintext state $\bar{s}_n = (\mathsf{S}, \mathsf{T}, \mathsf{W}, \gamma)$ to compute the final (encrypted) challenge state s_n as specified by the scheme. We bound the transition between the games by invoking the following lemma, which is implied by a reduction to the robustness of SE. We give the full proof in Appendix C.3.

Lemma 4.5.1 *Let (p, τ) be an error setting with associated transcript generator $\mathcal{T}$, and let $\Pi = (\mathsf{Reg}, \mathsf{Chk})$ be TypTop's password checker, with associated plaintext checker $\mathsf{PChecker}[\Pi]$. Let Π be implemented using the canonical PBE scheme $\mathsf{PBE}[\mathsf{SH}, \mathsf{SE}] = (\overline{\mathsf{E}}, \overline{\mathsf{D}})$ where SE is a symmetric encryption scheme and SH is a random oracle. Then for any adversary $\mathcal{A}$ running in time T and making at most q queries to SH, there exist adversaries $\mathcal{A}'$, $\mathcal{R}$ such that*

$$\mathsf{Adv}^{\text{offdist}}_{\Pi,\mathcal{T}}(\mathcal{A}) \leq \mathsf{Adv}^{\overline{\text{offdist}}}_{\Pi,\mathcal{T}}(\mathcal{A}') + 2 \cdot \mathsf{Adv}^{\text{rob}}_{\mathsf{SE}}(\mathcal{R}) + \frac{(t \cdot (n+1)+1)^2}{2^{(\kappa-1)}} ,$$

and, moreover, $\mathcal{A}'$ and $\mathcal{R}$ run in time $T' \approx T$, and $\mathcal{A}'$ makes at most q queries to SH. Here t denotes the cache size, SE takes as keys uniform bit strings of length κ, and n denotes the length of the transcript.

We now state our main theorem in which we upper bound the advantage of an attacker $\mathcal{A}$ in game $\text{OFFDIST}_{\Pi,\mathcal{T}}$.

Theorem 4.5.1 *Let (p, τ) be an error setting with associated transcript generator $\mathcal{T}$, and let $\Pi = (\mathsf{Reg}, \mathsf{Chk})$ be TypTop's password checker with associated plaintext checker $\mathsf{PChecker}[\Pi]$. Let Π be implemented using the canonical PBE scheme $\mathsf{PBE}[\mathsf{SH}, \mathsf{SE}] = (\overline{\mathsf{E}}, \overline{\mathsf{D}})$ where SE is a symmetric encryption scheme and SH is a random oracle. Then for any adversary $\mathcal{A}$ running in time T and making at most*

$\underline{\mathsf{PChecker}[\Pi](w_0, \tilde{w}_1, \ldots, \tilde{w}_n)}$
$\gamma \leftarrow\!\!\$\ \mathbb{Z}_\omega$; $(\mathsf{S}_0, \mathcal{U}_0) \leftarrow\!\!\$\ \mathsf{CacheInit}(w_0)$; $\mathsf{T}[0] \leftarrow w_0$
For $(\tilde{w}, i) \in \mathcal{U}_0$ do $\mathsf{T}[i] \leftarrow \tilde{w}$
For $k = 1, \ldots, n$ do
  $b \leftarrow 0$
  For $i = 0, \ldots, t$ do
    If $\tilde{w}_k = \mathsf{T}[i]$ then
      $b \leftarrow 1$; $\pi \leftarrow\!\!\$\ \mathsf{Perm}(\omega)$
      $(\mathsf{S}_k, \mathcal{U}_k) \leftarrow\!\!\$\ \mathsf{CacheUpdt}(\pi, \mathsf{S}_{k-1}, (\tilde{w}_k, i), \mathsf{W}[1], \ldots, \mathsf{W}[\omega])$
      For $(\tilde{w}', j) \in \mathcal{U}_k$ do $\mathsf{T}[j] \leftarrow \tilde{w}'$
      For $j = 1, \ldots, t$ do $\mathsf{T}'[\pi[j]] \leftarrow \mathsf{T}[j]$
      For $j = 1, \ldots, \omega$ do $\mathsf{W}[j] \leftarrow \varepsilon$
  If $b = 0$ then $\mathsf{W}[\gamma] \leftarrow \tilde{w}_k$; $\gamma \leftarrow \gamma + 1 \bmod \omega$
$\bar{s}_n \leftarrow (\mathsf{S}, \mathsf{T}, \mathsf{W}, \gamma)$
Return $\bar{s}_n$

Figure 4.5: The plaintext transcript checking scheme associated to Π with caching scheme $\mathsf{Cache} = (\mathsf{CacheInit}, \mathsf{CacheUpdt})$. All entries of tables T and W are initially set to $\perp$ and ε, respectively.

q queries to SH*, there exist adversaries* $\mathcal{B}, \mathcal{C}, \mathcal{R}, \mathcal{G}$ *such that*

$$\mathsf{Adv}^{\mathrm{offdist}}_{\Pi,\mathcal{T}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathrm{offguess}}_{\Pi,\mathcal{T}}(\mathcal{G}, q) + 2 \cdot \mathsf{Adv}^{\mathrm{mkror}}_{\mathsf{SE}}(\mathcal{B}, t) + \frac{(t+1)^2}{2^{\ell_{\mathrm{salt}}}} + 2 \cdot \mathsf{Adv}^{\mathrm{rob}}_{\mathsf{SE}}(\mathcal{R}) + 2 \cdot \mathsf{Adv}^{\mathrm{ror}}_{\mathsf{PKE}}(\mathcal{C}) + \frac{(t \cdot (n+1) + 1)^2}{2^{(\kappa-1)}} \ ,$$

and, moreover, $\mathcal{B}, \mathcal{C}, \mathcal{R}, \mathcal{G}$ *run in time* $T' \approx T$*. Here* t *denotes the cache size,* SE *takes as keys uniform bit strings of length* κ*, and* n *denotes the length of the transcript. The salts used to derive keys for the canonical PBE scheme are of length* ℓ_{salt}*. Adversary* $\mathcal{B}$ *makes* t *queries to its encryption oracle, and* $\mathcal{C}$ *makes* $\omega + 1$ *queries to its encryption oracle, where* ω *is the length of the wait list.*

The above theorem shows that the distinguishing advantage of an attacker in game $\mathrm{OFFDIST}_{\Pi,\mathcal{T}}$ is upper bounded by the probability that they can guess either the real password or a cached typo (which comprises the first term of the right hand side of the above equation) plus the remaining terms which, for the appropriate choice of cryptographic components and key sizes, can be assumed to be negligibly small. This implies that an attacker who compromises the state of the adaptive

checker in an offline attack learns no information about the underlying password and the login pattern, *unless* they can guess one of the cached values. We will analyze the probability that this occurs in Section 4.5.2.

We sketch the proof here and defer a detailed treatment to Appendix C.3. We first apply Lemma 4.5.1 to transition to game $\overline{\text{OFFDIST}}_{\Pi,\mathcal{T}}$ and thereby introduce the plaintext checker. We then argue by a series of game hops, beginning with the cache state s_n as in game $\overline{\text{OFFDIST}}_{\Pi,\mathcal{T}}$ with challenge bit $b = 0$. We then modify the PBE scheme to sample salts without replacement; this will ultimately be used to ensure that the attacker has to submit guesses to distinct cache positions when we reduce to game $\text{OFFGUESS}_{\Pi,\mathcal{T}}$. With this in place, we further set a flag which is set only if the attacker queries one of the cached values to his random oracle, allowing us to eventually reduce to the success probability of $\text{OFFGUESS}_{\Pi,\mathcal{T}}$. Finally we use the real-or-random ciphertext security of the SE and PKE schemes to replace all of the real ciphertexts in the state of the checker with random ciphertexts, thus transforming the state into a random one as per game $\overline{\text{OFFDIST}}_{\Pi,\mathcal{T}}$ with challenge bit $b = 1$.

On the use of PBKDFs. Above in our analysis we have abstracted away the details of the slow hash SH and modeled it simply as a random oracle. This allows accounting for queries to SH as unit cost, which will suffice for our analysis. One can go further, however, replacing SH with a true password-based key derivation function and converting the unit cost to that of computing the hash function (e.g., the cost of c applications of a standard hash function, in the case that SH is replaced by a hash chain construction such as PKCS#5). See for example [35, 97, 96] for a discussion of relevant results.

4.5.2 Security Against Offline Guessing Attacks

Having reduced the offline security of TypTop to the guessing game OFFGUESS, we now upper bound the success probability of an attacker in this game. Recall that p denotes the distribution of passwords chosen by users, $\mathcal{P}$ denotes the support of p, and τ denotes the family of typo distributions for each password $w \in \mathcal{P}$.

When TypTop is used in a given error setting, the probability that a typo lies in the cache of a given password depends inherently on the caching policy in use. For an error setting and instantiation of TypTop, we let $\tilde{\tau}$ denote the *induced cache inclusion function* where $\tilde{\tau}_w(\tilde{w})$ denotes the probability that $\tilde{w}$ is included in the typo cache of password w. We provide a general security analysis of TypTop in terms of $\tilde{\tau}$, then concretize our analysis by empirically modeling $\tilde{\tau}$ for a real world error setting and a number of caching policies. Letting $T[j]$ denote the distribution of the typo at position j in the cache, the fact that the set of cached typos are distinct and randomly permuted implies that $\mathbf{Pr}[T[j] = \tilde{w} \mid T[0] = w] = \frac{1}{t} \cdot \tilde{\tau}_w(\tilde{w})$, for all $0 < j \leq t$.

We would like to establish a class of typo distributions for which adding typo-tolerance via TypTop offers no security degradation over an exact checker. Since an exact checker accepts the correct password only, the analogous guessing game has the adversary attempting to guess a user's password with a budget of q guesses, and we denote the success probability achieved by an optimal attacker as λ_q. It is easy to see that the attacker's best strategy is to guess the q most probable passwords according to the distribution, so it follows that $\lambda_q = \sum_{i=1}^{q} p(w_i)$, where $w_1, w_2, \ldots$ denote the passwords in $\mathcal{P}$ sorted in decreasing order of their probability.

We define the *edge-weight* of a typo $\tilde{w}$ under the induced cache inclusion func-

tion $\tilde{\tau}$ to be $b_{\tilde{\tau}}(\tilde{w}) = \sum_{w \in \mathcal{P}} \tilde{\tau}_w(\tilde{w})$. Notice that for a given typo $\tilde{w}$, we have that $\tilde{\tau}_w(\tilde{w}) \in [0, 1]$ for *each* password w, so in theory the edge-weight could be very large. We say that an error setting is t-*sparse* with respect to TypTop with cache size t and a particular caching scheme, if for all $\tilde{w} \in \mathcal{M}$ it holds that $b_{\tilde{\tau}}(\tilde{w}) \leq t$. In the following theorem, we show that if an error setting is t-sparse then there is *no speedup* in an optimal offline attack against TypTop as opposed to an optimal offline attack against an exact checker ExChk.

Theorem 4.5.2 *Let (p, τ) be an error setting with associated transcript generator $\mathcal{T}$, and let $\Pi = (\mathsf{Reg}, \mathsf{Chk})$ be TypTop's password checker with typo cache size t. Then if the error setting is t-sparse with respect to Π, then for any adversary $\mathcal{G}$, it holds that*

$$\mathsf{Adv}^{\text{offguess}}_{\Pi,\mathcal{T}}(\mathcal{G}, q) \leq \lambda_q \quad \textit{where} \quad \lambda_q = \sum_{i=1}^{q} p(w_i)$$

The full proof is given in Appendix C.4; we provide a brief sketch here. We begin by splitting the set of guess / index pairs output by $\mathcal{G}$ into two exclusive sets — a set Z_0 consisting of guesses at the real password in cache position $\mathsf{T}[0]$, and a set Z_1 consisting of guesses at the cached typos in positions $\mathsf{T}[j]$ where $0 < j \leq t$. The success probability induced by the former set is simply $\sum_{w \in Z_0} p(w)$. The success probability contributed by the latter set is the expectation — over all passwords *not* already accounted for by the guesses at the real password in Z_0 — that one of these cache typo guesses succeeds. We show — via a general result which allows us to succinctly upper bound a certain class of summations in which this latter success probability lies — that the t-sparsity of the error setting implies that the guessing advantage arising from both sets is upper-bounded by λ_q, as required.

Are real world error settings t-sparse? It remains to establish whether real

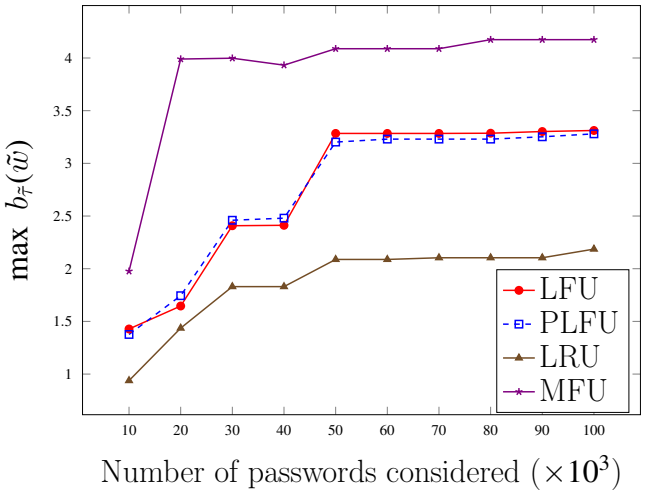


Figure 4.6: The change in the maximum edge-weight for different number of passwords considered from RockYou leak.

world error settings are t-sparse; if so, then Theorem 4.5.2 shows that we can enjoy the typo-tolerance offered by TypTop with *no loss* in offline security compared to an exact checker. It is easy to see that not every error setting is t-sparse; e.g., imagine (p, τ) such that all passwords are mistyped to the same typo, with no other typos possible. This one typo will lie in the cache of every password with probability close to 1, greatly degrading security and pushing the maximum edge-weight well above t.

In this section we use simulations to show that real world error settings (p, τ) are indeed t-sparse. We model the password distribution p using password data observed in the RockYou password leak [81], which consists of 14 million unique passwords from 32 million users. We sanitized the Rockyou data by removing passwords longer than 50 characters (as these are unlikely to be human chosen passwords) and shorter than 6 characters (as per common password policy recommendation), and set p equal to the resulting distribution. We model the typo distribution τ using data on user's password typing habits gathered via an MTurk experiment (see Section 4.6.1), and the data released with [30]. We describe how we built this model in Appendix C.2.

To compute the induced cache inclusion function $\tilde{\tau}$ for (p, τ) and TypTop with a given caching policy, we simulate the associated transcript generator $\mathcal{T}$ using the password distribution p and the typo distribution τ. For each password w, we sample n strings $\tilde{w}_1, \ldots, \tilde{w}_n$ according to τ_w. This captures a user with registered password w who makes a series of login attempts including some typos. We run PChecker (with default parameters) on input $(w, \tilde{w}_1, \ldots, \tilde{w}_n, w)$, where the final correct password entry is included to ensure at least one cache update occurs. We ran the simulation m times for each password w, and set $\tilde{\tau}_w(\tilde{w})$ to be the fraction of these m runs that the typo $\tilde{w}$ lies in the final cache of the password w. We repeat this process for all four caching policies described in Section 4.4. Via preliminary simulations on a small subset of randomly sampled passwords with different values of m and n, we found that $\tilde{\tau}_w$ changed very little for $n \geq 1000$ and $m \geq 200$; therefore we set $n = 1000$ and $m = 200$ for our full simulation.

Running the simulation for each of the passwords in the RockYou leak would be very slow, so we estimate $\tilde{\tau}_w$ using the k most probable passwords in the RockYou leak for $k \in \{1, \ldots, 10\} \times 10^4$. With this in place, we compute the edge-weight $b_{\tilde{\tau}}(\tilde{w}) = \sum_w \tilde{\tau}_w(\tilde{w})$ for each possible typo $\tilde{w}$, and report the maximum observed edge-weight for each k and caching policy in Figure 4.6.

For all four caching policies, we found that $\tilde{\tau}_w$ comfortably satisfies the desired 5-sparsity condition (recall that we implement TypTop with cache size $t = 5$) for all $k \leq 10^5$. The largest observed edge-weight was 4.2 for the MFU caching policy, with a maximum edge-weight of only 3.2 for the PLFU policy which we shall ultimately choose for deployment (see Section 4.6).

Moreover, we find that the maximum edge-weight increases minimally in the range $5 \times 10^4 \leq k \leq 10^5$ for all caching policies considered. This, coupled with the

fact that the maximum edge-weights are well below the required threshold of 5.0 for all k considered, indicates that if we were able to perform simulations on the entire password distribution, we would still find the error setting to be 5-sparse as required.

The top 10^5 passwords in the RockYou leak share some structural similarities (e.g., 90% of these passwords contain only letters and numbers, and only 5% are more than 10 characters in length). To check that these similarities are not biasing our results, and to gain further support for our conclusion that the simulated error setting (p, τ) is 5-sparse, we repeat the above experiment using two different sets of passwords to estimate $\tilde{\tau}_w$ and the corresponding edge-weights. The first set consists of k passwords chosen randomly from the support of p, and the second consists of the k most probable passwords when the top one million passwords are excluded from consideration. As before, we consider all values of $k \in \{1, \ldots, 10\} \times 10^4$.

We observed similar trends to those displayed in Figure 4.6, except that in these two experiments the maximum observed edge-weights are even better (in terms of security) at 2.8 and 3.0 respectively for the MFU caching scheme, and less for all other caching policies. This is likely to be because passwords which are distinct in structure induce more diverse sets of typos. This results in fewer typos being shared between multiple passwords, which in turn decreases their observed edge-weights.

To assess the benefit of the admissible typo policy described in Section 4.4, we additionally repeated the above experiments with these restrictions removed one at a time. We found that the error setting (p, τ) no longer remains 5-sparse if any of the three restrictions are removed. These simulations emphasize the importance of the admissible typo policy for security.

4.5.3 Security Against Online Attacks

We briefly discuss TypTop's resistance to online guessing attacks, in which an attacker attempts to impersonate a user via the login API of the system. The main difference between online and offline attacks against TypTop is that in the former, each guess the attacker makes is checked against *every* entry in the cache, whereas in the latter it is only checked against the specific cache slot guessed. We provide full details of these notions and our analyses in Appendix C.5

To estimate the decrease in the online security of TypTop compared to an exact checker, we approximate the success probability of an optimal online attacker using a greedy algorithm similar to that used by Chatterjee et. al. in [30], and data from real world password / typo distributions. We show that security loss is minimal for all caching schemes (namely a loss of 0.2% for the MFU policy, and less than 0.1% for all others, including the PLFU caching scheme which we ultimately choose for deployment). We also describe a simple blacklisting strategy, in which a small subset of the typos most beneficial to an attacker are excluded from the typo cache, and prove that this reduces the security loss to zero.

Another variety of online attack against adaptive checkers considers an adversary who is able to interleave his guesses with correct password submissions by the legitimate user. For TypTop these correct submissions trigger cache updates, so it is possible that one of the attacker's guesses, stored in the wait list, is allowed into the cache.

Recall that we set TypTop's admissible typo policy so only typos within DL distance one of the real password are allowed in the cache. This in itself provides a degree of protection against these interleaving attacks. We additionally propose

a simple countermeasure which virtually eliminates them. We can associate an origin tag (such as an IP in the web login setting, or TTY id in an SSH login) to each entry in the wait list; only entries originating from destinations at which the user has successfully authenticated are allowed to enter the cache. This may exclude the occasional typo made by a legitimate user from a new location; we defer a detailed treatment of this attack and countermeasures to future work.

4.6 Evaluating Utility

In this section, we investigate how successful TypTop is at correcting user's password typos under different parameter settings and ultimately select the best performing ones for real world deployment. To gather data about user's password typing patterns, we conducted an experiment on Amazon Mechanical Turk (MTurk).

We define the *utility* of a typo-tolerant PBAS to be the fraction of typos accepted by the checker taken across the population; formally the utility of a PBAS Π is defined $\mathsf{Utility}(\Pi)$ = # (typos accepted by Π) / # (typos observed). The same utility metric is used by Chatterjee et al. in [30], allowing for a direct comparison of results.

4.6.1 Data Collection From MTurk

Our study — designed to capture the password typing behavior of a user who first registers a password with a service, and then reenters this password to authenticate himself at subsequent logins — asks an MTurk worker to choose a password and type it repeatedly over a period of time. The design is similar to that of Komanduri

et al. in [109], and has two stages: registration and login. Registration consists of a single MTurk HIT (Human Intelligence Task) in which we ask the worker to choose a password that is at least 8 characters long, as if they were registering with an email provider. They are encouraged to choose a password which is strong and distinct from any of their existing passwords, and are informed that they must memorize this password for future logins. As with many registration forms, they must type the password twice. The user then fills in a short survey (1–2 questions) as means of distraction, and then is instructed to attempt to login with their registered password via a login form. The worker has to type the password correctly to be able to submit the HIT. An option for a "forgotten password" link is provided; however this link delays the HIT by 20 seconds, discouraging them from clicking it frequently. We informed workers that they must type their passwords manually and avoid browser auto-fills. We recorded every keypress made inside the password boxes and used heuristics to reject any submissions that did not appear to have been typed in.

After this registration step is complete, we create a sequence of 50 user-specific HITs for the worker, which they alone may view and complete; this allows us to track individual user's typing habits over different login attempts, while keeping their chosen password confidential. Each HIT consists of the same login form used in the final step of the registration stage. The user must complete the login, then answer a few questions on their demographics or password typing behavior. New HITs are created an hour after the submission of the last HIT to prevent workers from completing them back-to-back; we notify the workers when a new HIT is available. The workers are paid $0.05 for each HIT they complete, with an

additional bonus of $0.04 for completing every five HITs.

Data cleansing and demographics. Due to various incompatibility issues with rare browser versions and submissions which appeared to have been auto-filled, data from 42 workers had to be discarded. Of the remaining 438 workers, 271 (61.9%) made at least 10 login attempts. In all subsequent analysis, we only consider the data from these 271 workers. Based on the demographics survey, we found that 48% of the 271 workers are males and 52% female; only 35 (13%) of the users are left handed with the rest right handed. 117 (43%) of the users belong to the age group 18–30, 108 (40%) belong to the age group 31–45, 30 (11%) to the age group 46–60, and 16 (6%) are above the age of 60.

4.6.2 Analysis of Passwords and Typos

All of the passwords chosen by the MTurk workers were unique, with a median and average length of 10 and 10.9 characters respectively, and average estimated zxcvbn strength of 31.5 bits. Most passwords included special characters, numbers and different case letters. Overall the passwords chosen by MTurk workers were stronger than those seen in most password leaks (e.g., passwords in the RockYou leak have a median password length of 7 characters and average zxcvbn entropy of 14.2 bits). While we expect our results to hold for weaker password choices, further studies will need to be performed to confirm this.

The workers made a median of 30 login attempts per person and 8,739 login attempts in total; 491 (5.6%) of these contained at least one incorrect password submission. Within the login attempts, there were a total of 9,440 password submissions; of these 701 (7.4%) were incorrect, with 484 (70%) of these incorrect

Typo Category	% of Typos	% of Users
Caps Lock	14	21
Shift first char	4	7
One insertion	12	28
One deletion	12	25
One replacement	31	47
Transposition	4	7
Two insertions	3	5
Two deletions	3	5
Two replacements	10	20
Other	8	16

Figure 4.7: Categorization of typos observed in the MTurk study. The middle column gives the percentage of observed typos which were of each category. The rightmost column gives the percentage of users who made a typo of that category.

submissions lying within DL distance 2 of the real password — as per the discussion in Section 5.2, we classify these 484 incorrect submissions as typos, and therefore eligible for typo-tolerance in our subsequent analysis. A classification of typos into exclusive categories is shown in Figure 4.7; our study finds similar user typo behavior to that observed by Chatterjee et al. in [30]. Looking ahead, we will calculate the utility of a given implementation of TypTop to be the fraction of these 484 typos accepted by that implementation.

A graph depicting the DL distances of incorrect password submissions is given in Figure 4.9a (the blue line). In total, 366 separate login attempts (4.2%) required at least one password resubmission due to a typo. We found that 167 users (62%) made at least one typo, with 95 (35%) making at least two typos in two different login attempts.

4.6.3 Simulation Setup

We perform simulations using the data from our MTurk study to evaluate the utility of various combinations of cache sizes, caching schemes, and typo admission policies. In more detail, we consider:

- **Cache size:** While a larger cache allows us to accept more typos and increases utility, this benefit needs to be weighed up against the greater computational power required to process larger caches, and the potential degradation in online security from allowing more typos to authenticate. We consider caches of size t for $t \in \{2, 3, 5\}$ in our simulations.
- **Caching Schemes:** We consider all of the caching schemes discussed in Section 4.4: LRU, LFU, PLFU, MFU, and Best-t. To see how TypTop compares with the relaxed checking approach of Chatterjee et al., we also run simulations for the relaxed checker using the Top-t corrector functions as per [30], which in order of their efficacy are: flipping the case of all characters, flipping the case of the first character, removing the last character, removing the first character, and applying shift to switch the last digit to a symbol.
- **Typo admission policy**: As discussed in Section 4.4, admissible typos must satisfy three criteria:
  (1) $\mathrm{DL}(w, \tilde{w}) \leq d$; (2) $\mu_{\tilde{w}} > m$; and (3) $\mu_{\tilde{w}} > \mu_w - \sigma$. We investigated all combinations of the values $d \in \{1, 2\}$, $m \in [0, 40]$, and $\sigma \in [0, 9]$.
- **Warming up the cache:** Since warming up the cache allows us to tolerate typos from the very first login, it can only increase utility. Additionally our security simulations in Section 4.5, which were all performed with warmed caches, indicate that this practice does not negatively impact security. As such we warm caches with the t most probable typos according to our typo model (see

CP	$d = 1$			$d = 2$		
	t=2	t=3	t=5	t=2	t=3	t=5
LFU	18	21	26	19	24	31
PLFU	19	22	27	22	26	32
MFU	18	21	26	19	25	30
LRU	17	21	27	19	25	32
Best-t	16	19	23	16	19	23
Top-t	18	19	22	18	19	22

Figure 4.8: The utility of different caching policies (CP), cache sizes (t) and edit distance cutoff (d) for admissible typos, when applied to the login transcripts of all MTurk workers who made at least one typo. We impose no guessability restrictions on admissible typos.

Appendix C.2).

For each set of parameter choices in the scope of the above, we simulate each worker's login behavior by replaying their password submissions to the password checker being evaluated. We report the utility for each of these configurations below; recall that the utility of a checker Π is defined to be Utility(Π) = # (typos accepted by Π) / # (typos observed), where both numerator and denominator are taken across all users.

4.6.4 Results

Caching policies and cache sizes. We first compare the efficacy of different caching schemes, cache sizes t, and DL distance thresholds d. For these simulations, we impose no guessability restrictions on admissible typos, setting $m = 0$ and $\sigma = \infty$, as this will maximize the number of admissible typos. The utility of each strategy is shown in Figure 4.8. We see that larger cache sizes (which allow us to cache and tolerate more typos) correspond to an increase in utility. Similarly increasing the DL distance threshold (and with it the set of typos considered for inclusion in the cache) also increases utility. However, we conjecture that allowing

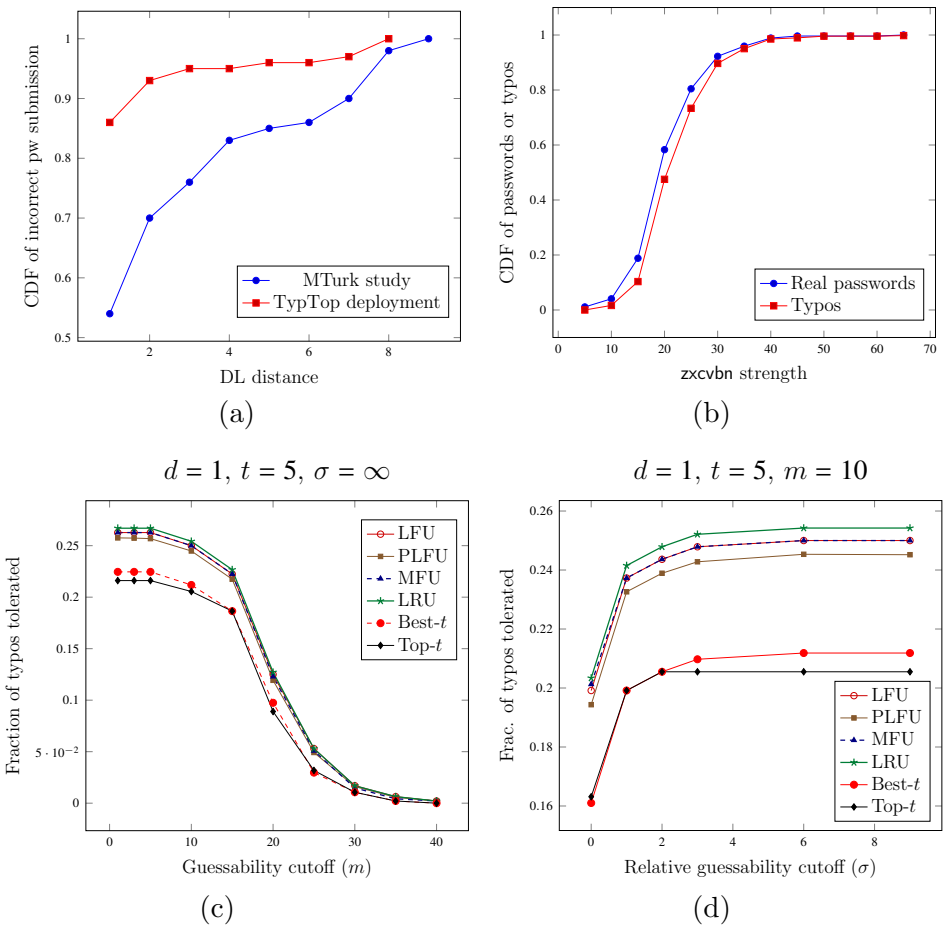


Figure 4.9: (a) CDF of fraction of incorrect password submissions within a given DL distance of the real password. (b) CDFs of zxcvbn strength of real passwords compared to typos in the MTurk study. (c),(d) The change in utility for different absolute guessability thresholds (m) and relative guessability thresholds (σ). The values of other parameters are specified above each chart.

typos with DL distance 2 will degrade security enough to outweigh the utility benefits (e.g., the resulting error settings are less likely to be t-sparse), so we select $t = 5$ and $d = 1$ for further analysis and deployment.

For caching schemes, perhaps unsurprisingly, the utility of relaxed checking with Top-t correctors is very similar to that of the static policy Best-t — the former performs better for $t = 2$, while the latter performs better for $t = 5$. This is because in our unoptimized typo model, the common error of flipping

the case of the first character often does not appear among the two most probable typos for a given password, whereas it does get corrected by Top-2 correctors; the resulting typos missed by Best-2 allows Top-2 to outperform it. However for $t = 5$, the benefit of the password-specific typo correction offered by Best-5 emerges, catching typos that the Top-5 correctors (which are chosen based on population-wide analysis) cannot correct. With more data about typos and future refinement of the parameters should overcome this small difference.

Because each individual worker made only a small number of incorrect submissions in our study, the choice of caching policy had little observable impact on utility for cache sizes $t \geq 3$. However the variation is more noticeable for cache size $t = 2$. As anticipated LRU performs less well than the frequency based caching policies. Among the latter set, PLFU performs the best. Surprisingly, we also see that MFU underperforms PLFU; however this could be due to the fact that we did not receive enough incorrect submissions to see the benefit of MFU emerge, and conjecture that in a longer term study, MFU may outperform PLFU.

Due to strong performance for utility and security (as discussed in Section 4.5), and the cache size restraints, (which make MFU unsuitable for practical purposes), we choose PLFU as the caching policy for deployment, with a cache size of $t = 5$ and DL distance threshold $d = 1$.

Different guessability restrictions. Next we investigate the impact of different admissible typo parameters m and σ on utility (recall that for typo $\tilde{w}$ to be considered for inclusion in the cache of password w, it must be the case that $\mu_{\tilde{w}} > m$, and $\mu_{\tilde{w}} > \mu_w - \sigma$). We begin by considering the guessability cutoff parameter $m \in \{0, \ldots, 40\}$. We perform simulations for TypTop implemented with cache size $t = 5$, edit distance threshold $d = 1$, but without any relative guessabil-

ity cutoff imposed ($\sigma = \infty$). We compute utility for all guessability cutoff values $m \in \{0, \ldots, 40\}$ and all caching policies; the results are shown in Figure 4.9c. As expected, utility decreases as the guessability cutoff increases and fewer typos are considered for inclusion in the cache. Notably, we see the utility decrease rapidly for $m > 10$. This is because, as shown in Figure 4.9b, nearly 5% of the observed typos have an estimated security strength of less than 10 bits, indicating that setting $m = 10$ gives the maximum security benefit without significantly degrading utility.

We then investigated the effect on utility of different relative guessability cutoff parameter settings σ. We perform simulations with the same parameter settings as above, except we now fix $m = 10$. We compute utility for all relative guessability cutoff parameters $\sigma \in [0, 9]$ and all caching policies; the results are shown in Figure 4.9c. For most of the caching policies, we found no significant improvement on utility for $\sigma > 3$. This is to be expected, given that the guessability of passwords and typos are very similar (as shown in Figure 4.9b). Since increasing the guessability cutoff further increases the possibility that a significantly more guessable string enters the cache with minimal benefit to utility, we choose $\sigma = 3$ to optimize the balance between utility and security.

Users and logins benefited. Among the 167 users who made at least one typo during their submission (recall, here a typo is an incorrect submission within edit distance 2 of the real password), 75 users (44.9%) would benefit by having at least one of their typos accepted by TypTop with the PLFU caching policy and the parameter setting described above. In contrast, only 49 users (29.3%) would receive this benefit from the relaxed checker of [30] implemented with the Top-5 correctors. Of the 366 login attempts containing at least one typo, 106 (29%) of

these would require at least one less password resubmission if TypTop were used as the password checker. This saves an average of 5 seconds per login attempt for the users making at least one typo, and an average of 12 seconds for the users who made two typos in two different logins. In our small MTurk study (271 users), we find that TypTop would save 23 person-minutes of login time, and with larger user populations expect to see this time saved grow to several person-months.

We test the null hypothesis that TypTop (with the parameters decided above) does not outperform relaxed checking with Top-t correctors in tolerating typos for a randomly chosen user using a Wilcoxon signed-rank test [110]. We found that we can reject the null hypothesis with p-value < 0.001.

4.7 A case study with TypTop

Password authenticated personal device logins are a key scenario in which users may benefit from the typo-tolerance offered by TypTop. In this section we discuss how to build TypTop for typo-tolerant device logins in Mac OS X and Linux based systems. We deploy this prototype on 25 volunteers' machines, obtaining data on their password typing behaviors, and report on the performance of TypTop in a real world deployment.

Implementation of TypTop. We build TypTop as a pluggable authentication module (PAM) [111] for Unix based systems. The state of TypTop for each user is stored in a separate file with the same permissions as the `/etc/shadow` file, which is used to store users' password hashes and is readable only by the root. The `Chk` procedure is implemented in C++ and installed with similar file permissions to `passwd`, a Linux utility tool which is used to update a user's password. Via

modifying the PAM configuration files, we set TypTop to be engaged on almost all applications which require password based authentication. TypTop is very simple to install, but requires root privileges.

As detailed in Section 4.6, we implement TypTop with parameters: $\{\mathsf{CP}$=PLFU, t=5, d=1, m=10, σ=3$\}$. There is currently no option to customize the parameters; we plan to introduce restricted control for system admins in the next version.

We tested the computational overhead of TypTop in an Ubuntu 14.04 laptop with Intel Core M processor and 8 GB of memory. The size of the state-file is 13KB. The average turnaround time is 110 milliseconds for a successful authentication (i.e., the correct password or a cached typo is entered), and 250 milliseconds for an incorrect submission. For traditional Ubuntu logins (e.g., with `su`), login takes less than 1 millisecond.

The main computational overhead incurred by TypTop is due to our use PBKDF2 [35] with 20,000 iterations of SHA-256 for each computation of the hash function SH, while (somewhat surprisingly) a default Ubuntu laptop uses only 1 iteration of SHA-256 for its hash computation. Current standards [35, 112] recommend using at least 5,000 iterations of SHA-256 to hash passwords.

Chk will always perform the maximum number of hashes for an incorrect submission, which is why we see a longer turn around time for failed authentications. However, this overhead is well below the noticeable limit for users. To avoid timing side channels, we might want to compute the maximum number of hash computations for every login attempt (successful or failed) to make these turnaround times

constant.

Pilot deployment of TypTop. To gather data on TypTop's efficacy, we modify the implementation of TypTop slightly to allow confidential logging of users' password typing behavior.

The logging module works as follows. During password registration, two random 16-byte strings are generated and stored in the state of TypTop; the first is used as an HMAC key to compute a unique identifier for each submitted password (or typo), while the second is used as a user-device identifier. With every password submission, we log the HMAC identifier of the submission, along with the DL distance from the real password, the relative guessability, and whether the typo could have been corrected using Top-5 correctors (to form a comparison between TypTop and relaxed checking). This allows us to learn the transcript of password and typo submissions — and thus simulate a run of TypTop — while never learning the actual strings entered. Some of these values (e.g., DL distance) require both the underlying password and the typo to process; we back-fill these values after a successful authentication when the encrypted caching scheme state and wait list are decrypted. To simulate the Best-5 static caching scheme, we also log the identifiers of the five most probable typos of the password.

The log and corresponding user identifier are uploaded to a server via an HTTPS post request every 10 logins, after which the uploaded log is deleted from the user's laptop. The key used to generate the HMAC identifiers is never uploaded, making it impossible to brute-force recover passwords given the information uploaded. Users can disable the logging and / or uploading feature at any

time during the study without any effect on the functionality.

Data collection and analysis. We initially advertised our study via two university mailing lists and a number of social media groups. However, we received a lower response rate than anticipated, as users were initially reluctant to run research code as their primary mode of authentication. Therefore, while not optimal, we used snowball sampling [113] to increase participation in our study.

Study participation is anonymous: we do not know the exact set of volunteers who installed TypTop, although we can deduce the number of distinct machines on which TypTop was installed from the logs generated. In subsequent analysis, we assume that each user installed TypTop on only one machine. We collected data from 25 users over a period of 22–100 days (different users have varying data collection periods depending on when they joined the study, and how long they chose to participate for). TypTop was used for a median of 27 days.

We observed a total of 4,563 password submissions during the study, with a median of 103 submissions per user. The average user types their password more than five times a day, and incorrectly 15% of the time. We found that 93% of these incorrect submissions are within DL distance two of the real password and thus are classified as typos. The fraction of incorrect submissions that constitute typos is larger than that observed in the MTurk study (70%); this is probably because MTurk workers created their 'passwords' for the study, and so were more likely to make high DL distance errors due to misremembering them.

In total we observed 501 typos, of which 316 (63%) are tolerated by TypTop. This is significantly higher than the 122 (22%) that would be tolerated by the relaxed checker with Top-5 correctors. However the benefit of typo correction

varies across users. We observed 2 users who received an especially great benefit from TypTop. They used TypTop for over 45 days, during which they typoed their password 24% of the time — we found TypTop corrected 85% of their typos, whereas the Top-5 corrector functions corrected virtually none. For the remainder of the participants, we found that TypTop and the Top-5 correctors performed roughly the same. We believe this is because personal machines in university settings tend to have very lenient password policies (if any policy at all), meaning users pick simpler passwords whose typos are more likely to be among those corrected by the Top-5 correctors. We expect to see the performance benefit of TypTop emerge in settings where users must pick stronger and less readily correctable passwords (e.g., over 12 characters), and lock / unlock their computers many times a day. We simulated other caching policies on the collected data, and observed roughly the same performance.

These results suggest that personalized typo-tolerance may be very beneficial to users who are especially typo-prone, and that this benefit increases the longer the system is used. The sample size of our initial pilot deployment is too small to allow us to draw more general conclusions at this stage; as such we are planning a study with more participants and which runs for a longer period of time. We will be publishing TypTop as a public, open-source project to facilitate a study with a broader set of participants.

4.8 Conclusion

We introduce the notion of personalized typo-tolerant password checkers, which adapt over time to correct the typos made most frequently by the individual user.

We design and build an adaptive password checking scheme called TypTop, which securely caches incorrect password submissions that pass a policy check on what forms of typos to allow. We formalize a cryptographic security notion for such schemes, and show a formal reduction of our scheme's security to the difficulty of brute-force cracking attacks against the registered password or the typos entered into the typo cache. We give simple criteria that, if met, ensure no security loss in offline attack and negligible security loss in online attack, and empirically verify that real world password and typo distributions satisfy this requirement. Simulations conducted with data gathered via a study on Amazon MTurk suggest that TypTop will outperform existing approaches to typo-tolerant password checking, and a small pilot deployment suggests that TypTop can provide a substantial usability benefit to especially typo-prone users.

CHAPTER 5

PROTOCOLS FOR CHECKING COMPROMISED CREDENTIALS

This chapter is under submission to ACM Conference on Computer and Communication Security (CCS), 2019 [32].

5.1 Introduction

Password database breaches have become routine [114]. Such breaches enable credential stuffing attacks, in which attackers try to compromise accounts by submitting passwords that were leaked from another website. To counter credential stuffing attacks, companies and other organizations have begun checking if their users' passwords appear in breaches, and, if so, they deploy further protections (e.g., resetting the user's password or otherwise warning the user). Information on what usernames and passwords have appeared in breaches is gathered either from public sources or from a third-party service. The latter democratizes access to leaked credentials, making it easy for others to help their customers gain confidence that they are not using exposed passwords. We refer to such services as *compromised credentials checking* services, or C3 services in short.

Two prominent C3 services already operate. HaveIBeenPwned (HIBP) [115] was deployed by CloudFlare in 2018 and is used by many web services, including Firefox [116], EVE Online [117], and 1Password [118]. Google released a Chrome extension called Password Checkup (GPC) [2] in February 2019 that allows users to check if their username-password pair appears in a compromised dataset. Both services work by having the user share with the C3 server a prefix of the hash of their password or the hash of their username-password pair. This leaks some

information about user passwords, which is problematic should the C3 server be compromised or otherwise malicious. But until now there has no thorough investigation into the damage from the leakage of current C3 services or suggestions for protocols that provide better privacy.

We provide the first formal treatment of C3 services for different settings, including exploration of their security requirements. A C3 service must provide secrecy of credentials provided by the client, and ideally, it should also preserve secrecy of the leaked datasets held by the C3 server. The computational and bandwidth overhead for the client and especially the server should also be low. The server might hold billions of leaked records, so using existing cryptographic protocols for private set intersection (PSI) [119] to check if the client's credentials are present in the server's dataset will be prohibitively expensive.

Current industry-deployed C3 services therefore reduce bandwidth requirements by dividing the leaked data into buckets before performing PSI. The client shares with the C3 server the identifier of the bucket where their credentials would be found, if present in the dataset. Then, the client and the server engage in a private set intersection protocol between the bucket held by the server and the credential held by the client. In current schemes, the prefix of the hash of the user credential is used as the bucket identifier. The client shares the hash prefix (bucket identifier) of their credentials with the C3 server.

Revealing hash prefixes of the credentials may be dangerous. We outline an attack scenario against such prefix-revealing C3 services. In particular, we consider a conservative setting where an attacker obtains the hash prefix shared with the C3 server (possibly by compromising the server) and also knows the username associated with the queried credential. We rigorously evaluate the security of

HIBP and GPC under this threat model via a mixture of formal and empirical analysis.

We start by considering users with a password appearing in some leak and show how to adapt a recent state-of-the-art credential tweaking attack [120] to take advantage of the knowledge of hash prefixes. In a credential tweaking attack, one uses the leaked password to determine likely guesses (usually, small tweaks on the leaked password). Via simulation, we show that our variant of credential tweaking successfully compromises 80% of such accounts within 1,000 guesses, given the transcript of a query made to the HIBP server. This is 28% more than what an attacker who does not have the hash prefix and simply runs the best known credential tweaking attack would get.

We also consider user accounts not present in a leak. Here we found that the leakage from the hash prefix disproportionately affects security compared to the previous case. For these user accounts, obtaining the query to HIBP enables the attacker to guess 71% of passwords within 1,000 guesses, which is a 12x increase over the success with no hash prefix information. Similarly, for GPC, our simulation shows 34% of user passwords can be guessed in 10 or fewer attempts (and 61% in 1,000 attempts), should the attacker learn the hash prefix shared with the GPC server.

The attack scenarios described are conservative because they assume the attacker can infer which queries to the C3 server are associated to which usernames. This may not be always possible. Nevertheless, caution dictates that we would prefer schemes that leak less. We therefore present two new C3 protocols, one that checks for leaked passwords (like HIBP) and one that checks for leaked username-password pairs (like GPC). Like GPC and HIBP, we *partition* the password space

before performing PIR, but we do so in a way that reduces leakage significantly.

Our first scheme works when only passwords are queried. It utilizes a novel approach that we call frequency-smoothing bucketization (FSB). The key idea is to use an estimate of the distribution of human-chosen passwords to assign passwords to buckets in a way that flattens the distribution of accessed buckets. We show how to obtain good estimates (using leaked data), and, via simulation, that FSB reduces leakage significantly. In many cases the best attack given the information leaked by the C3 protocol works no better than not having the information at all. While the benefits come with some added computational complexity and bandwidth, we show via experimentation that the operational overhead for the FSB C3 server or client is comparable with the overhead from GPC, while also leaking much less information.

We also present a more secure bucketizing scheme that provides better privacy/bandwidth tradeoff for C3 servers that store username-password pairs. We give a simple modification of the protocol used by GPC. We call this IDB, ID-based bucketization, as it uses the hash prefix of only the user identifier for bucketization (instead of the hash prefix of the username-password pair as used by GPC). Not having password information in the bucket identifier hides the user's password perfectly from an attacker who obtains the client queries (assuming that passwords are independent of usernames). We implement IDB and show that the average bucket size in this setting for a hash prefix of 16 bits is similar to that of GPC (around 9,166 entries per bucket).

Contributions. The main contributions of this chapter are the following:

- We provide a formalization of C3 protocols and detail the security goals for

such services.

- We discuss various threat models for C3 services, and analyze the security of two widely deployed C3 protocols. We show that an attacker that learns the queries from a client can severely damage the security of the client's passwords, should they also know the client's username.
- We give a new C3 protocol for checking only leaked passwords (FSB) that utilizes the knowledge of the human-chosen password distribution to reduce the leakage.
- We give a new C3 protocol for checking leaked username-password pairs (IDB) that bucketizes using only usernames.
- We analyze the performance and security of both new C3 protocols to show feasibility in practice.

We will release as public, open source code our server and client implementations of FSB and IDB.

5.2 Overview

We investigate approaches to checking credentials present in previous breaches. Several third party services provide credential checking, enabling users and companies to mitigate credential stuffing and credential tweaking attacks [120, 121, 122], an increasingly daunting problem for account security.

To date, such C3 services have not received any analysis, and indeed their design rationale has only been discussed in blog posts [1, 123]. We start by describing the architecture of such services, and then we detail relevant threat models.

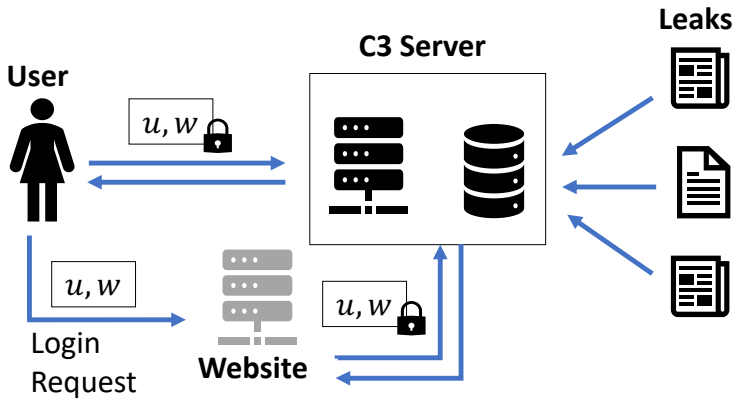


Figure 5.1: A C3S service allows a client to ascertain whether a username and password appear in public breaches known to the service.

5.2.1 C3 settings.

We provide a diagrammatic summary of the abstract architecture of C3 services in Figure 5.1. A C3 *server* has access to a breach database $\tilde{\mathcal{S}}$. We can think of $\tilde{\mathcal{S}}$ as a set of size N, which consists of either a set of passwords $\{w_1, \ldots, w_N\}$ or username-password pairs $\{(u_1, w_1), \ldots, (u_N, w_N)\}$. This corresponds to two types of C3 services — *password-only C3 service* and *username-password C3 service*. For example, HIBP [1] is a password-only C3 service,[1] and Google's service GPC [2] is an example of username-password C3 service.

A *client* has as input a credential $s = (u, w)$ and wants to determine if s is at risk due to exposure. The client and server therefore engage in a set membership protocol to determine if $s \in \tilde{\mathcal{S}}$. Here, clients can be users themselves (query C3 service using, say, a browser extension), or other web services can query the C3 service on behalf of their users. Of course, clients may make multiple queries to the C3 service, though the number of queries might be rate limited.

[1]Actually HIBP also allows checking if a user identifier (email) is leaked with a data breach. For the purpose of this study, however, we only focus on the above mentioned two C3 services.

The ubiquity of breaches means that, nowadays, the breach database $\tilde{\mathcal{S}}$ will be quite large. A recently leaked compilation of previous breached data contains 1.4 billion username password pairs [124]. The HIBP database has 501 million unique passwords [1]. Google's blog specifies that there are 4 billion username-password pairs in their database of leaked credentials [123].

C3S protocols should be able to scale to handle set membership requests for these huge datasets and for millions of requests a day. HIBP reported serving around 600,000 requests per day on average [125]. The design of C3S should therefore not be computationally expensive on the server-side. The number of network round trips required must be low, and we will restrict attention to protocols that can be completed with a single HTTPS request. Finally, we will want to minimize bandwidth usage.

5.2.2 Threat model.

Both the C3 server's database $\tilde{\mathcal{S}}$ and the client's queried password should be considered confidential. While breaches are often made public, we prefer to treat $\tilde{\mathcal{S}}$ as confidential even if it consists of only public information. Of course, by querying on $\tilde{\mathcal{S}}$ a malicious client will fundamentally be able to check if values are in the database. Ideally such a brute-force approach would be the best possible attack.

A malicious C3 server could deviate from its protocol, for example, by lying to the client about the contents of $\tilde{\mathcal{S}}$ in order to encourage them to pick a weak password. Monitoring techniques might be useful to catch such misdeeds. We do not consider active attacks further, as we focus instead on the more pressing issue of not leaking (u, w) to an honest-but-curious server that follows its protocol but

wants to infer information about the user's username or password.

An attacker upon compromising the C3 server can observe the query metadata (e.g., IP address of the querying user). An attacker can learn the username corresponding to a query by linking IP addresses to usernames using auxiliary information. For example, the attacker can send tracking emails to all the leaked email addresses present in the breach datasets, and if a client clicks on the link present in the email, the attacker would be able to retrieve the IP address of the user [126]. Thereby, the attacker can associate the email to queries. Such attack might be avoidable by using anonymity networks like Tor [127]. Of course, ideally, a C3 protocol should to not leak the username to the server.

Nevertheless, for the rest of the chapter, we will focus on the threat model where the attacker knows the querying user's username, and refer to it as a known-username attack (KUA). The KUA is targeted; as such, the attacker can take advantage of the leaked data to find (any) leaked passwords associated to the target username and tailor its guesses based on them.

For this chapter, we will focus on online attack settings, where the attacker tries to impersonate a user by guessing their password for other web services online. These are easy to launch and are one of the most prevalent forms of attacks [14, 28]. However, in an online setting, the web service can monitor the failed login attempts and lock an account out after too many incorrect password submissions. Therefore, the attacker gets only a small number of attempts, known as the guessing budget q of the attack.

Potential approaches. A C3 protocol requires, at core, a secure set membership query. Existing protocols for private set intersection (a generalization of

Credentials checked	Name	Bucket identifier	B/w (KB)	RTL (ms)	Security loss
Password	HIBP	20-bits of SHA1(w)	15.9	208	12x
	FSB	Figure 5.6, $\bar{q} = 10^2$	261	527	2x
(Username, password)	GPC	16-bits of Argon2($u \| w$)	606	458	10x
	IDB	16-bits of Argon2(u)	606	487	1x

Figure 5.2: Comparison of different C3 protocols. HIBP [1] and GPC [2] are two C3 services used in practice. We introduce frequency-smoothing bucketization (FSB) and identifier-based bucketization (IDB). Security loss is computed assuming query budget $q = 10^3$ for users who has not been compromised before.

set membership) [128, 129, 130, 131] cannot currently scale to the set sizes required in C3 settings, $N \approx 2^{30}$. For example, the basic PSI protocol that uses an oblivious pseudorandom function (OPRF) [128] computes $y_i = F_K(u_i, w_i)$ for $(u_i, w_i) \in \tilde{\mathcal{S}}$ where F_K is the secure OPRF with secret key K (held by the server). It sends all $y_1, \ldots, y_N$ to the client, and the client obtains $y = F_K(u, w)$ for its input (u, w) by obliviously computing it with the server. The client can then check if $y \in \{y_1, \ldots, y_N\}$. But clearly for large N this is prohibitively expensive in terms of bandwidth. One can use Bloom filters to more compactly represent the set $y_1, \ldots, y_N$, but the result is still too large. While more advanced PSI protocols exist that improve on these results asymptotically, they are unfortunately not yet practical for this C3 setting [128, 119].

Practical C3 schemes therefore relax the security requirements, allowing the protocol to leak some information about the client's queried (u, w) but hopefully not too much. To date no one has investigated how damaging the leakage of currently proposed schemes is, which we turn to doing next. In Figure 5.2, we show all the different settings for C3 we discuss in the chapter, and compare their security and performance.

Symbol	Description
u / $\mathcal{U}$	user identifier, e.g. email / domain of users
w / $\mathcal{P}$	password / domain of passwords
$\mathcal{S}$	domain of credentials
$\tilde{\mathcal{S}}$	set of leaked credentials, and $\lvert\tilde{\mathcal{S}}\rvert = N$
p	distribution of username-password pairs over $\mathcal{U} \times \mathcal{P}$
p_w	distribution of passwords over $\mathcal{P}$
$\hat{p}_s$	estimate of p_w used by C3 server
q	query budget of an attacker
$\bar{q}$	parameter to FSB, estimated query budget of an attack

Figure 5.3: Descriptions of the notation used in the chapter.

5.3 Bucketization Schemes and Security Models

In this section we formalize the security models for a class of C3 schemes that bucketize the breach dataset into smaller sets (buckets). Intuitively, a straightforward approach for checking whether or not a client's credentials are present in a large set of leaked credentials hosted by a server is to divide the leaked data into various buckets. The client and server can then perform a private set intersection between the user's credentials and one of the buckets (potentially) containing that credential. The bucketization makes private set membership tractable, while only leaking to the server that the password may lie in the set associated to a certain bucket.

We give a general framework to understand the security loss and bandwidth overhead of different bucketization schemes, and evaluate existing C3 services for the same.

Notation. For ease of description of the constructions that follow, we fix some notations. Let $\mathcal{P}$ be the set of all passwords, and p_w be the associated probability distribution; let $\mathcal{U}$ be the set of all user identifiers, and p be the joint distribution over $\mathcal{U} \times \mathcal{P}$. We will use $\mathcal{S}$ to denote the domain of credentials being checked,

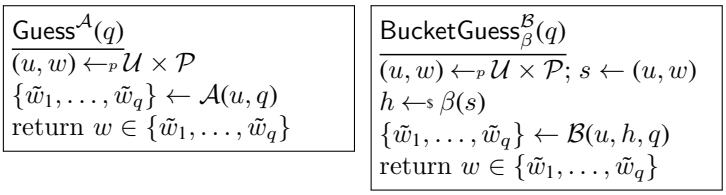


Figure 5.4: The guessing games to evaluate security of different C3 schemes.

i.e., for password-only C3 service, $\mathcal{S} = \mathcal{P}$, and for username-password C3 service, $\mathcal{S} = \mathcal{U} \times \mathcal{P}$. Below we will use $\mathcal{S}$ to give a generic scheme, and specify the setting only if necessary to distinguish. Similarly, $s \in \mathcal{S}$ denotes a password or a username-password pair, based on the setting. Let $\tilde{\mathcal{S}}$ be the set of leaked credentials, and $|\tilde{\mathcal{S}}| = N$.

Let H be a cryptographic hash function from $\{0,1\}^* \mapsto \{0,1\}^{\ell}$, where ℓ is a parameter to the system. We use $\mathcal{B}$ to denote the set of buckets, and we let $\beta\colon \mathcal{S} \mapsto \mathcal{B}^* \setminus \{\varnothing\}$ be a bucketizing function which maps a credential to a set of buckets. A credential can be mapped to multiple buckets, and every credential is assigned to at least one bucket. An inverse function to β is $\alpha\colon \mathcal{B} \mapsto \mathcal{S}^*$, which maps a bucket to the set of all credentials it contains; so, $\alpha(b) = \{s \in \mathcal{S} \mid b \in \beta(s)\}$. Note, $\alpha(b)$ can be very large given it considers all credentials in $\mathcal{S}$. We let $\tilde{\alpha}$ be the function that denotes the credentials in the buckets held by the C3 server, $\tilde{\alpha}(b) = \alpha(b) \cap \tilde{\mathcal{S}}$.

The client sends b to the server, and then the client and the server engage in a set intersection protocol between $\{s\}$ and $\tilde{\alpha}(b)$.

5.3.1 Bucketization schemes.

Bucketization is dividing the credentials held by the server into smaller buckets. The client can use the bucketizing function β to find the set of buckets for a credential, and then pick one randomly to query the server. There are different ways to bucketize the credentials.

In the first method, which we call hash-prefix-based bucketization (HPB), the credentials are partitioned based on the first l bits of a cryptographic hash of the credentials. GPC [2] and HIBP [1] APIs use HPB. The distribution of the credentials is not considered in HPB, which causes it to incur higher security loss, as we show in Section 5.4.

We introduce a new bucketizing method, which we call frequency-smoothing bucketization (FSB), that takes into account the distribution of the credentials and replicates credentials into multiple buckets if necessary. The replication "flattens" the conditional distribution of passwords given a bucket id, and therefore vastly reduces the security loss. We discuss FSB in more details in Section 5.5.

In both HPB and FSB, the bucketization function depends on the user's password. We give another bucketization approach — the most secure one — that bucketizes based only on the hash prefix of the user identifier. We call this identifier-based bucketizing (IDB). The approach is only applicable for username-password C3 services. We discuss IDB in Section 5.4.

5.3.2 Security measure.

The goal of an attacker is to learn the user's password. We will focus on online-guessing attacks, where an attacker tries to guess a user's password over the login interfaces provided by a web service. An account might be locked for too many incorrect guesses (say, for example, 10), and the attack fails. Therefore, we will measure an attacker's success given a certain guessing budget, say q. We will always assume the attacker has access to the username of the target user.

The security games are given in Figure 5.4. The game Guess models the situation in which no information besides the username is revealed to the adversary about the password. In the game $\mathsf{BucketGuess}$, the adversary also gets access to a bucket that is chosen according to the credentials $s = (u, w)$ and the bucketization function β.

We define the advantage against a game as the maximum probability that the game outputs 1. Therefore,

$$\mathsf{Adv}^{\mathsf{gs}}(q) = \max_{\mathcal{A}} \Pr\left[\, \mathsf{Guess}^{\mathcal{A}}(q) \Rightarrow 1 \,\right],$$

and

$$\mathsf{Adv}^{\mathsf{b\text{-}gs}}_{\beta}(q) = \max_{\mathcal{B}} \Pr\left[\, \mathsf{BucketGuess}^{\mathcal{B}}_{\beta}(q) \Rightarrow 1 \,\right].$$

The probabilities are taken over the choices of username-password pairs and the selection of bucket from the bucketizing function β. Security loss, $\Delta_{\beta}(q)$, of a bucketizing protocol β is defined as the ratio of $\mathsf{Adv}^{\mathsf{b\text{-}gs}}_{\beta}(q)$ over $\mathsf{Adv}^{\mathsf{gs}}(q)$.

Note,

$$\Pr\left[\, \mathsf{Guess}^{\mathcal{A}}(q) \Rightarrow 1 \,\right] = \sum_{u} \Pr\left[\, w \in \mathcal{A}(u, q) \wedge U = u \,\right].$$

To maximize this probability, the attacker must pick the q most probable passwords

for each user. Therefore,

$$\mathsf{Adv}^{\mathsf{gs}}(q) = \sum_{u} \max_{w_1,\ldots,w_q} \sum_{i=1}^{q} \Pr[\, W = w_i \wedge U = u \,] \,. \tag{5.1}$$

In $\mathsf{BucketGuess}_\beta$, the attacker has access to the bucket identifier, and therefore the advantage is computed as

$$\begin{aligned}
&\mathsf{Adv}_{\beta}^{\mathsf{b\text{-}gs}}(q) \\
&= \sum_{u} \sum_{b} \max_{w_1,\ldots,w_q} \sum_{i=1}^{q} \Pr[\, W = w_i \wedge U = u \wedge B = b \,] \\
&= \sum_{u} \sum_{b} \max_{\substack{(u,w_1),\ldots,(u,w_q) \\ \in \alpha(b)}} \sum_{i=1}^{q} \frac{\Pr[\, W = w_i \wedge U = u \,]}{|\beta((u,w_i))|}
\end{aligned}$$

The second equation follows because for $b \in \beta((u,w))$, each bucket in $\beta(w)$ is equally likely to be chosen, so

$$\Pr[\, B = b \mid W = w \wedge U = u \,] = \frac{1}{|\beta((u,w))|} \,.$$

The joint distribution of usernames and passwords is hard to model. To simplify the equations, we divide the users targeted by the attacker into two groups: *compromised* (users whose previously compromised accounts are available to the attacker) and *uncompromised* (users for which the attacker has no information other than their usernames).

We assume the there is no direct correlation between the username and password.[2] Therefore, an attacker cannot use the knowledge of only the username to tailor guesses. This means that in the uncompromised setting, we assume $\Pr[\, W = w \mid U = u \,] = \Pr[\, W = w \,]$. Assuming independence of usernames and passwords, we define in the uncompromised setting

$$\lambda_q = \mathsf{Adv}^{\mathsf{gs}}(q) = \max_{w_1,\ldots,w_q} \sum_{i=1}^{q} \Pr[\, W = w_i \,] \,. \tag{5.2}$$

[2]Though prior work [122, 132] suggests knowledge of only username can improve efficacy of guessing user passwords, the improvement is minimal. See Appendix D.2 for more on this analysis.

We give analytical and empirical analysis of security in this setting, and show that the security of uncompromised users is impacted by existing C3 schemes much more than that of compromised users.

In the compromised setting, the attacker can use the username to find other leaked passwords associated with that user, which then can be used to tailor guesses [120, 122]. Analytical bounds on the compromised setting are less informative, so we evaluate this setting empirically in Section 5.6.

Bandwidth. The bandwidth required for a bucketization scheme is determined by the size of the buckets. The maximum size of the buckets can be determined using a balls-and-bins approach [133], assuming the client picks a bucket randomly from the possible set of buckets $\beta(s)$ for a credential s, and $\beta(s)$ also maps s to a random set of buckets. In total $m = \sum_{s \in \tilde{\mathcal{S}}} |\beta(s)|$ credentials (balls) are "thrown" into $n = |\mathcal{B}|$ buckets. If $m > |\mathcal{B}| \cdot \log |\mathcal{B}|$, then following the seminal results on balls-and-bins game [133], we can show the maximum number of passwords in a bucket with very high probability $1 - o(1)$ is less than $\frac{m}{n} \cdot \left(1 + \sqrt{\frac{n \log n}{m}}\right) \leq 2 \cdot \frac{m}{n}$. We will use this formula to compute an upper bound on the bandwidth requirement for specific bucketization schemes.

5.4 Hash-prefix-based Bucketization

Hash-prefix-based bucketization (HPB) schemes are a simple ways to divide the credentials stored by the C3 server. In this scheme, a prefix of the hash of the credential is used as the criteria to group the credentials into buckets — all credentials that share the same hash-prefix are assigned to the same bucket. The total number of buckets depends on l, the length the hash-prefix. The number of

credentials in the buckets depends on both l and $|\tilde{\mathcal{S}}|$. We will use $\mathsf{H}^{(l)}(\cdot)$ to denote the function that outputs the l-bit prefix of the hash $\mathsf{H}(\cdot)$. The client shares the hash prefix of the credential they wish to check with the server. While a smaller hash-prefix reveals less information to the server about the user's password, it also increases the size of each bucket held by the server, which in turn increases the communication overhead.

Hash-prefix-based bucketization is currently being used for credential checking in industry: HIBP [1] and GPC [2]. We introduce a new HPB protocol called IDB that achieves zero security loss for any query budget. Below we will discuss the design details of these three C3 protocols.

HIBP [1]. HIBP uses HPB bucketization to provide a password-only C3 service. They do not provide compromised username-password checking. HIBP maintains a database of leaked passwords, which contains more than 501 million passwords [1]. They use the SHA1 hash function, with prefix length $l = 20$; the leaked dataset is *partitioned* into 2^{20} buckets. The prefix length is chosen to ensure no bucket is too small or too big. With $l = 20$, the smallest bucket has 381 passwords, and the maximum bucket has 584 passwords [134] . This effectively makes the user's password k-anonymous. However, k-anonymity provides limited protection, as shown by numerous prior works [135, 136, 137] and by our security evaluation.

The passwords are hashed using SHA1 and indexed by their hash prefix for fast retrieval. A client computes the SHA1 hash of their password w and queries HIBP with the 20-bit prefix of the hash; the server responds with all the hashes that shares the same 20-bit prefix. The client then checks if the full SHA1 hash of w is present among the set of hashes sent by the server. This is a weak form of PSI that does not hide the leaked passwords from the client — the client learns

the SHA1 hash of the leaked passwords and can perform brute force cracking to recover those passwords.

HIBP justifies this design choice by observing that passwords in the server side leaked dataset are publicly available for download on the Internet. Therefore, HIBP lets anyone download the hashed passwords and usernames. This can be useful for parties who want to host their own leak checking service without relying on HIBP. However, keeping the leaked dataset up-to-date can be challenging, making a third-party C3 service preferable.

HIBP trades server side privacy for protocol simplicity. The protocol also allows utilization of heavy caching on content delivery networks (CDN), such as Cloudflare.[3] The caching helps HIBP to be able to serve 8 million requests a day with 99% cache hit rate (as of August 2018) [138]. The human-chosen password distribution is "heavy-headed", that is a small number of passwords are chosen by a large number of users. Therefore, a small number of passwords are queried a large number of times, which in turn makes CDN caching much more effective.

GPC [2]. Google provides a username-password C3S, called Password Checkup (GPC). The client — a browser extension — computes the hash of the username and password together using the Argon2 hash function with the first $l = 16$ bits to determine the bucket identifier. After determining the bucket, the client engages in a private set intersection (PSI) protocol with the server. The full algorithm is given in Figure 5.5. GPC uses an OPRF-based PSI protocol. Let $F_{(\cdot)}(\cdot)$ be a key-homomorphic pseudo-random function (PRF) such that $F_a(\cdot) \times F_b(\cdot) = F_{ab}(\cdot)$. Under the hood, F calls the hash function H on $u\|w$, and then maps the hash output onto the elliptic curve point for further computation.

[3] https://www.cloudflare.com/

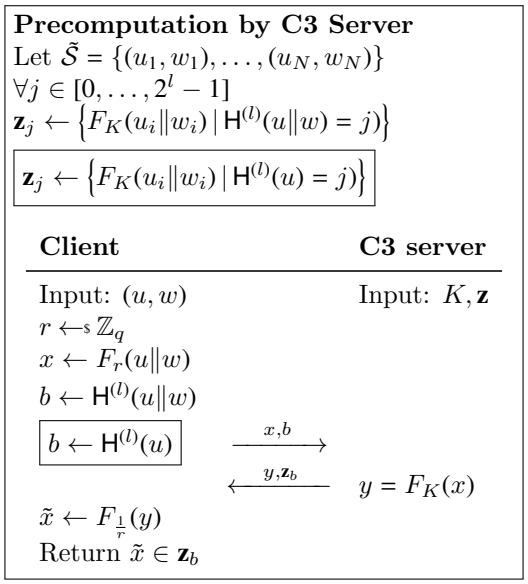


Figure 5.5: Algorithms for GPC, and the change in IDB given in the box. $F_{(\cdot)}(\cdot)$ is a PRF.

The server has a secret key K which it uses to compute the $y_i = F_K(u_i \| w_i)$. The client shares with the server the bucket id b and the PRF output of username-password pair $x = F_r(u \| w)$, for some randomly sampled r. The server returns the bucket $\mathbf{z}_b = \{y_i \mid \mathsf{H}(u_i \| w_i) = b\}$ and $y = F_K(x)$. Finally, the client completes the OPRF computation by computing $\tilde{x} = F_{\frac{1}{r}}(y) = F_K(u \| w)$, and checking if $\tilde{x} \in \mathbf{z}_b$.

The GPC protocol is significantly more complex than HIBP, and it does not allow easy caching by CDNs. However, it provides secrecy of server side leaked data — the best case attack is to follow the protocol to brute-force check if a password is present in the leak database.

Bandwidth. HPB assigns every credential to only one bucket, therefore, $m = \sum_{w \in \tilde{\mathcal{S}}} |\beta(w)| = |\tilde{\mathcal{S}}| = N$. The total number of buckets $n = 2^l$. Following the discussion from Section 5.3, maximum bandwidth for a HPB C3S should be no more than $2 \cdot \frac{m}{n} = 2 \cdot \frac{N}{2^l}$.

We experimentally verified the bandwidth value, and the sizes of the buckets for HIBP, GPC, and IDB are given in Section 5.7.

Security. HPB schemes like HIBP and GPC expose a prefix of the user's password (or username-password pair) to the server. As discussed earlier, we assume the attacker knows the username of the target user. In the uncompromised setting — where the user identifier does not appear in the leaked data available to the attacker, we show that giving the attacker the hash-prefix with a guessing budget of q queries is equivalent to giving as many as $q \cdot |\mathcal{B}|$ queries (with no hash-prefix) to the attacker.

Theorem 5.4.1 *Let $\beta_{\text{HPB}} : \mathcal{S} \mapsto \mathcal{B}$ be the bucketization scheme that, for a credential $s \in \mathcal{S}$, chooses a bucket that is a function of $\mathsf{H}^{(l)}(s)$, where s contains the user's password. The advantage of an attacker in this setting against previously uncompromised users is*

$$\mathsf{Adv}^{\mathsf{b\text{-}gs}}_{\beta_{\text{HPB}}}(q) \leq \mathsf{Adv}^{\mathsf{gs}}(q \cdot |\mathcal{B}|) \,.$$

Proof: First, note that $|\beta_{\text{HPB}}(\cdot)| = 1$, as every password is assigned to exactly one of the buckets. Following the discussion from Section 5.3, assuming independence of usernames and passwords in the uncompromised setting, we can compute the advantage against game $\mathsf{BucketGuess}$ as,

$$\mathsf{Adv}^{\mathsf{b\text{-}gs}}_{\beta_{\text{HPB}}}(q) = \sum_{b \in \mathcal{B}} \max_{\substack{w_1,\ldots,w_q \\ \in \alpha(b)}} \sum_{i=1}^{q} \Pr[\, W = w_i \,] \leq \mathsf{Adv}^{\mathsf{gs}}(q \cdot |\mathcal{B}|)$$

We relax the $\alpha(b)$ notation to denote set of passwords (instead of username-password pairs) assigned to a bucket b. The inequality follows from the fact that each password is present in only one bucket. If we sum up the probabilities of the

top q passwords in each bucket, the result will be at most the sum of the probabilities of the top $q \cdot |\mathcal{B}|$ passwords. Therefore, the maximum advantage achievable is $\mathsf{Adv}^{\mathsf{gs}}(q \cdot |\mathcal{B}|)$.

Theorem 5.4.1 only provides an upper bound on the security loss. Moreover, for the compromised setting, the analytical formula is less informative. So, we use empiricism to find the effective security loss against compromised and uncompromised users. We report all security simulation results in Section 5.6. Notably, with GPC with hash prefix length $l = 16$, an attacker can guess passwords of 60.5% of (previously uncompromised) user accounts in fewer than 1000 guesses, a 10x increase from the percent it can compromise without access to the hash prefix. (See Section 5.6 for more results.)

Identifier-based bucketization (IDB). As our security analysis and simulation show, the security degradation of HPB is dismal. The main issue with those protocols is that the bucket identifier is a deterministic function of the user password. We give a new C3 protocol that uses HPB style bucketing based on only username. We call this identifier-based bucketization (IDB). IDB is defined for username-password C3 schemes.

IDB is a slight modification of the protocol used by GPC— we use the hash-prefix of the username, $\mathsf{H}^{(l)}(u)$, instead of the hash-prefix of the username-password combination, $\mathsf{H}^{(l)}(u\|w)$, as a bucket identifier. The scheme is described in Figure 5.5, using the changes in the boxed code. The bucket identifier is computed completely independent of the password (assuming username is independent of the password). Therefore, the attacker gets no additional advantage for knowing the bucket identifier.

Because IDB uses the hash-prefix of the username as the bucket identifier, two hash computations are required on the client side for each query (as opposed to one for GPC). With most modern devices, this is not a significant computing burden, but the protocol latency may be impacted, since we use a slow hash (Argon2). We show experimentally how the extra hash computation affects the latency of IDB in Section 5.7.

Since IDB does not use the user's password to determine the bucket identifier, there is no security loss.

Theorem 5.4.2 *With the* IDB *protocol, for all* $q \geq 0$

$$\mathsf{Adv}^{\mathsf{b\text{-}gs}}_{\mathrm{IDB}}(q) = \mathsf{Adv}^{\mathsf{gs}}(q).$$

We provide the proof of this theorem in Appendix D.3. Because the bucket identifiers are chosen independent of the passwords, the conditional probability of the password given the bucket identifier remains the same as the probability without knowing the bucket identifier.

Overall, we can use a form of HPB to create a username-password C3S scheme with no security loss, but the password-only C3S schemes constructed using HPB lead to significant security loss. In the next section we solve this problem by introducing a more secure password-only C3S scheme.

5.5 Frequency-Smoothing Bucketization

In the previous section we show how to build a username-password C3 service that does not degrade security. However, many services, such as HIBP, only provide a password-only C3 service. HIBP does not store username-password pairs so, should the HIBP server ever get compromised, an attacker cannot use their leak database to mount credential stuffing attacks. Moreover, IDB cannot be extended in any useful way to protect password-only C3 services.

Therefore, we introduce a new bucketization scheme to build secure password-only C3 services. We call this scheme frequency-smoothing bucketization (FSB). FSB assigns a password to multiple buckets based on its probability — frequent passwords are assigned to many buckets. Replicating a password into multiple buckets effectively reduces the conditional probabilities of that password given a bucket identifier. We do so in a way that makes the conditional probabilities of popular passwords similar to those of unpopular passwords to make it harder for the attacker to guess the correct password. FSB, however, is only effective for non-uniform credential distributions, such as password distributions.[4] Therefore, FSB cannot be used to build a username-password C3 service.

Implementing FSB requires knowledge of the distribution of human-chosen passwords. Of course, obtaining precise knowledge of the password distribution can be difficult, therefore, we will use an estimated password distribution, denoted by $\hat{p}_s$. Another parameter of FSB is $\bar{q}$, which is an estimate of the attacker's query budget. We show that if the actual query budget $q \leq \bar{q}$, FSB has zero security loss. Larger $\bar{q}$ will provide better security; however, it also means more replication of the

[4]Usernames (e.g., emails) are unique for each users, so the distribution of usernames and username-password pairs are close to uniform.

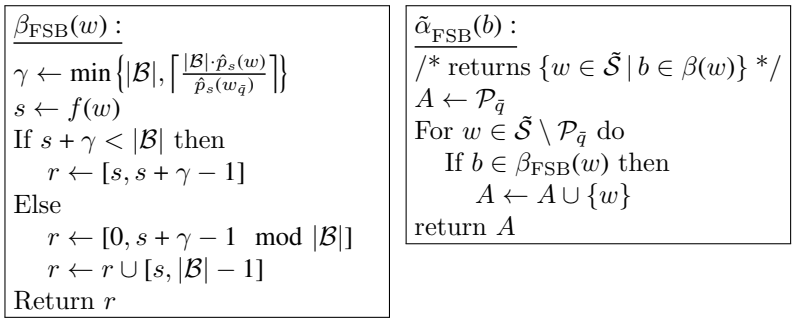


Figure 5.6: Bucketizing function β_{FSB} for assigning passwords to buckets in FSB. Here $\hat{p}_s$ is the distribution of passwords; $\mathcal{P}_{\bar{q}}$ is the set of top-$\bar{q}$ passwords according to $\hat{p}_s$; $\mathcal{B}$ is the set of buckets; f is a universal hash function $f\colon W \mapsto \mathbb{Z}_{|B|}$; $\tilde{\mathcal{S}}$ is the set of passwords hosted by the server.

passwords and larger bucket sizes. So, $\bar{q}$ can be tuned to balance between security and bandwidth. Below we will give the two main algorithms of FSB scheme: β_{FSB} and $\tilde{\alpha}_{\text{FSB}}$, followed by bandwidth and security analysis for FSB.

Bucketizing function (β_{FSB}). To map passwords to buckets, we use an universal hash function $f : \mathcal{P} \mapsto \mathbb{Z}_{|B|}$. The algorithm for bucketization $\beta_{\text{FSB}}(w)$ is given in Figure 5.6. The parameter $\bar{q}$ is used in the following way: β replicates the most probable $\bar{q}$ passwords, $\mathcal{P}_{\bar{q}}$, across all $|\mathcal{B}|$ buckets. Each of the remaining passwords are replicated proportional to their probability. A password w with probability $\hat{p}_s(w)$ is replicated exactly $\gamma = \left\lceil \frac{|\mathcal{B}|\cdot\hat{p}_s(w)}{\hat{p}_s(w_{\bar{q}})} \right\rceil$ times, where $w_{\bar{q}}$ is the $\bar{q}^{th}$ most likely password. Exactly which buckets a password is assigned to are determined using the universal hash function f. Each bucket is assigned an identifier between $[0, |\mathcal{B}|-1]$. A password w is assigned to the buckets whose identifiers fall in the range $[f(w), f(w)+\gamma-1]$. The range can wrap around. For example, if $f(w)+\gamma > |\mathcal{B}|$, then the password is assigned to the buckets in the range $[0, f(w)+\gamma-1 \mod |\mathcal{B}|]$ and $[f(w), |\mathcal{B}|-1]$.

Bucket retrieving function ($\tilde{\alpha}$). Retrieving passwords assigned to a bucket is challenging in FSB. An inefficient — linear in N — implementation of $\tilde{\alpha}$ is given

in Figure 5.6. Storing the contents of each bucket separately is not feasible, since the number of buckets in FSB can be very large, $|\mathcal{B}| \approx N$. To solve the problem, we utilize the structure of the bucketizing procedure where passwords are assigned to buckets in continuous intervals. This allows us to use an interval tree [139] data structure to store the intervals for all of the passwords. Interval trees allow fast queries to retrieve the set of intervals that contain a queried point (or interval) — exactly what is needed to instantiate $\tilde{\alpha}$.

This efficiency comes with increased storage cost. To store N entries in a interval tree, we require $\mathcal{O}(N \log N)$ storage. The tree can be built in $\mathcal{O}(N \log N)$ time, and each query takes $\mathcal{O}(\log N + |\tilde{\alpha}(b)|)$ time. The big-O notation only hides small constants.

Estimating password distributions. To construct the bucketization algorithm for FSB, the server needs an estimate of the password distribution (p_w). This estimate will be used by both the server and the client to assign passwords to buckets. One possible estimate is the histogram of the passwords in the leaked data $\tilde{\mathcal{S}}$. Histogram estimates are typically accurate for popular passwords, but such estimates are not complete — passwords that are not in the leaked dataset will have zero probability according to this estimate. Moreover, sending the histogram over to the client is expensive in terms of bandwidth and security critical. We also considered password strength meters, such as zxcvbn [103] as a proxy for a probability estimate. However, this estimate turned out to be too coarse for our purposes. For example, more than 10^5 passwords had a "probability" of greater than 10^{-3}.

We build a 3-gram password model $\hat{p}_n$ using the leaked passwords present in $\tilde{\mathcal{S}}$. Markov models or n-gram models are shown to be effective at estimating human-

chosen password distributions [41], and very fast to train and run (unlike neural network based password distribution estimators, such as [104]). However, we found the n-gram model assigns very low probabilities to popular passwords. The sum of the probabilities of the top 1,000 passwords as estimated by the 3-gram model is only 0.0012, whereas in practice the top 1000 passwords are chosen by 5.6% of users.

We therefore use a combined approach that uses a histogram model for the popular passwords and the 3-gram model for the rest of the distribution. Such combined techniques are also used in practice for password strength estimation [103, 104]. Let $\hat{p}_s$ be the estimated password distribution used by FSB. Let $\hat{p}_h$ be the distribution of passwords implied by the histogram of passwords present in $\tilde{\mathcal{S}}$. Let $\tilde{\mathcal{S}}_t$ be the set of the t most probable passwords according to $\hat{p}_h$. We used $t = 10^6$.

$$\hat{p}_s(w) = \begin{cases} \hat{p}_h(w) & \text{if } w \in \tilde{\mathcal{S}}_t \ , \\ \hat{p}_n(w) \cdot \frac{1-\sum_{\tilde{w}\in\tilde{\mathcal{S}}_t} \hat{p}_h(w)}{1-\sum_{\tilde{w}\in\tilde{\mathcal{S}}_t} \hat{p}_n(w)} & \text{otherwise.} \end{cases}$$

Bandwidth. We use the formulation provided in Section 5.3 to compute the bandwidth requirement for FSB. In this case, $m = |\mathcal{B}| \cdot \bar{q} + \frac{|\mathcal{B}|}{\hat{p}_s(w_{\bar{q}})} + N$, and $n = |\mathcal{B}|$. Therefore, the maximum size of a bucket is with high probability less than $2 \cdot \left(\bar{q} + \frac{1}{\hat{p}_s(w_{\bar{q}})} + \frac{N}{|\mathcal{B}|}\right)$. The details of this analysis are given in Appendix D.1.

In practice, we can choose the number of buckets to be such that $|\mathcal{B}| = N$. Then, the number of passwords in a bucket depends primarily on the parameter $\bar{q}$. Note, bucket size increases with $\bar{q}$.

Security analysis. We show that there is no security loss in the uncompromised setting for FSB when the actual number of guesses q is less than the parameter $\bar{q}$,

and we give an upper bound for the security loss when q exceeds $\bar{q}$.

Theorem 5.5.1 *If a frequency based bucketization scheme ensures* $\forall w \in \mathcal{P}$, $|\beta_{\text{FSB}}(w)| = \min\left\{|\mathcal{B}|, \left\lceil \frac{|\mathcal{B}| \cdot \hat{p}_s(w)}{\hat{p}_s(w_{\bar{q}})} \right\rceil\right\}$, *then for the uncompromised users,*

(1) $\mathsf{Adv}^{\text{b-gs}}_{\text{FSB}}(q) = \mathsf{Adv}^{\text{gs}}(q)$ *for* $q \leq \bar{q}$, *and*

(2) *for* $q > \bar{q}$,

$$\frac{\lambda_q - \lambda_{\bar{q}}}{2} \leq \Delta_q \leq (q - \bar{q}) \cdot \hat{p}_s(w_{\bar{q}}) - (\lambda_q - \lambda_{\bar{q}})$$

The full proof is included in Appendix D.4. Intuitively, since the top q passwords are repeated across all buckets, having a bucket identifier does not allow an attacker to easily guess these q passwords. Moreover, the conditional probability of these q passwords given the bucket is greater than that of any other password in the bucket. Therefore, the attacker's best choice is to guess the top q passwords, meaning that it does not get any additional advantage when $\bar{q} \leq q$, leading to part (1) of the theorem.

The proof of part (2) follows from the upper and lower bounds on the number of buckets each password beyond the top q is placed within. The bounds we prove show that the additional advantage in guessing the password in q queries is less than the number of additional queries times the probability of the $\bar{q}^{th}$ password and at least half the difference in the guessing probabilities λ_q and $\lambda_{\bar{q}}$ (defined in (5.2)).

Note that this analysis of security loss is based on the assumption that the FSB scheme has access to the precise password distribution, $\hat{p}_s = p_w$. We empirically

analyze the security loss in Section 5.6 for $\hat{p}_s \neq p_w$, in both the compromised and uncompromised settings.

5.6 Empirical Security Evaluation

In this section we empirically evaluate and compare the security loss for different password-only C3 schemes we have discussed so far — hash-prefix-based bucketization (HPB) and frequency-smoothing bucketization (FSB).

We focus on known-username attacks (KUA), since in many deployment settings a curious (or compromised) C3 server can figure out the username of the querying user. We separate our analysis into two settings: previously *compromised* users, where the attacker has access to one or more existing passwords of the target user, and previously *uncompromised* users, where no password corresponding to the user is known to the attacker (or present in the breached data).

Recall, according to our threat model, we assume the adversary has knowledge of all the leak dataset C3 is using. This situation is realistic, since many password breaches are readily available for download online. For each setting the attacker also knows the bucketizing algorithm. The attacker obtains (possibly by compromising the C3 service) the bucket identifier that the client queried, as well as the user's username or email. Our analysis will also show what an honest-but-curious C3 server would learn about the passwords of a user who participated in the protocol.

First we will look into the unrestricted setting where no password policy is enforced, and the attacker and the C3 server have the same amount of information

	$\tilde{\mathcal{S}}$	T	$T \cap \tilde{\mathcal{S}}$	T_{sp}	$T_{\text{sp}} \cap \tilde{\mathcal{S}}$
# users	383.2	7.5	5.6 (76%)	4.8	3.7 (77%)
# passwords	255.2	5.4	3.6 (67%)	4.0	2.4 (60%)
# user-pw pairs	748.9	7.5	2.8 (37%)	4.9	1.8 (37%)

Figure 5.7: Number of entries (in millions) in the breach dataset $\tilde{\mathcal{S}}$, test dataset T, and the site-policy test subset T_{sp}. Also reported the intersection (of users, passwords, and user-password pairs, separately) between the test dataset entries and the whole breach dataset that the attacker has access to. The percentage values refer to the fraction of the values in each test set that also appear in the intersections.

about the password distribution. In the second experiment, we analyze the effect on security of giving the attacker more information compared to the C3 server (defender) by having a password policy that the attacker is aware of but the C3 server is not.

Password breach dataset. We used the breach dataset used in [120]. The dataset was derived from a previous breach compilation [124] dataset containing about 1.4 billion username-password pairs. The data was cleaned by removing non-ASCII characters and passwords longer than 30 characters. The authors of [120] also joined accounts with similar usernames and passwords using a method they called the *mixed method*. The usernames with only one email and password were removed, which in total removed 650 million username-password pairs. We obtained this joined and filtered dataset from the authors and performed our empirical analysis on that dataset. Removal of the 650 million pairs for users with only one password can only affect the experiment on the security for uncompromised users. Given the large size of the dataset, we expect our results on attack success are not impacted in any significant way by the removal of those accounts.

The final dataset consists of about 756 million username-password pairs.[5] We remove 1% of username-password pairs to use as test data, denoted as T. The

[5]Note, there are duplicate username-password pairs in this dataset.

remaining 99% of the data is used to simulate the database of leaked credentials $\tilde{\mathcal{S}}$. For the experiments with an enforced password policy, we took the username-password pairs in T that met the requirements of the password policy to create T_{sp}. We use T_{sp} to simulate queries from a website which only allows passwords that are at least 8 characters long and are not present in Twitter's list of banned passwords [102]. For all attack simulations, the target user-password pairs are sampled from the test dataset T (or T_{sp}).

In Figure 5.7, we report some statistics about T, T_{sp}, and $\tilde{\mathcal{S}}$. Notably, 5.6 million (76%) of the users in T are also present in $\tilde{\mathcal{S}}$. This is likely because users in the joined breach compilation dataset have at least two passwords. If the 650 million singleton users had not been removed, we expect that this number would be smaller. Among the username-password pairs, 2.8 million (37%) pairs in T are also present in $\tilde{\mathcal{S}}$. This means an attacker will be able to compromise 37% of the accounts (which is 50% of the previously compromised accounts) trivially with credential stuffing. In the site-policy enforced test data T_{sp}, a similar proportion of the users (77%) and username-password pairs (37%) are also present in $\tilde{\mathcal{S}}$.

Experiment setup. We want to understand the impact of revealing a bucket identifier on the security of uncompromised and compromised users separately. As we can see from Figure 5.7, a large proportion of users in T are also present in $\tilde{\mathcal{S}}$. We therefore split T into two parts: one with only username-password pairs from compromised users, T_{comp} (users with at least one password present in $\tilde{\mathcal{S}}$), and another with only pairs from uncompromised users T_{uncomp}. We take two sets of random samples of 5000 username-password pairs[6], one from T_{comp}, and another from T_{uncomp}. For each pair (u, w), we run the games Guess and

[6]There was a low standard deviation between results for different random samples of 5000 pairs.

BucketGuess as specified in Figure 5.4. We record the results for guessing budgets of $q \in \{1, 10, 10^2, 10^3\}$. We repeat each of the experiments 5 times and report the averages in Figure 5.8.

For HPB, we compared implementations using hash prefixes of lengths $l \in \{12, 16, 20\}$. We use the SHA256 hash function with a salt, though the choice of hash function does not have a noticeable impact on the results.

For FSB, we used interval tree data structures to store the leaked passwords in $\tilde{\mathcal{S}}$ for fast retrieval of $\tilde{\alpha}(b)$. We used $|\mathcal{B}| = 2^{30}$ buckets and the hash function f is set to $f(x) = \mathsf{H}^{(30)}(x)$, the 30-bit prefix of the (salted) SHA256 hash of the password.

Attack strategy. The attacker's goal is to maximize its success in winning the games Guess and BucketGuess. In (5.1) and (5.2) we outline the advantage of attackers against Guess and BucketGuess, and thereby specify the best strategies for attacks. Guess denotes the baseline attack success rate in a scenario where the attacker does not have access to bucket identifiers corresponding to users' passwords. Therefore the best strategy for the attacker $\mathcal{A}$ is to output the q most probable passwords according to its best knowledge of the password distribution.

The optimal attack strategy for $\mathcal{B}$ in BucketGuess will be to find a list of passwords according to the following equation,

$$\underset{\substack{w_1,\ldots,w_q \\ b \in \beta((u,w_i))}}{\operatorname{argmax}} \sum_{i=1}^{q} \frac{\Pr[\, W = w_i \mid U = u \,]}{|\beta((u, w_i))|},$$

where the bucket id b and user identifier u are provided to the attacker. This is equivalent to taking the top-q passwords in the set $\alpha(b)$ ordered by $\Pr[\, W = w \mid U = u \,]/|\beta((u, w))|$.

We compute the list of guesses outputted by the attacker for a user u and bucket b in the following way. For the compromised users, i.e., if $(u, \cdot) \in \tilde{\mathcal{S}}$, the attacker first considers the list of 10^4 targeted guesses generated based on the credential tweaking attack introduced in [120]. If any of these passwords belong to $\alpha(b)$ they are guessed first. This step is skipped for uncompromised users.

For the remaining guesses, we first construct a list of candidates L. L consists of 10^6 most frequent passwords in $\tilde{\mathcal{S}}$ and 500×10^6 passwords generated from the 3-gram password distribution model $\hat{p}_n$. Each password w in L is assigned a weight $\hat{p}_s(w)/|\beta((u, w))|$ (See Section 5.5 for details on $\hat{p}_s$ and $\hat{p}_n$). The list L is pruned to only contain unique guesses. Note L is constructed independent of the username or bucket identifier, and it is reordered based on the weight values. Therefore, it is constructed once for each bucketization strategy. Finally, based on the bucket identifier b, the remaining guesses are chosen from $\{\alpha(b) \cap (u, w) \mid w \in L\}$ in descending order of weight.

For the HPB implementation, each password is mapped to one bucket, so $|\beta(w)| = 1$ for all w. For FSB, $|\beta(\cdot)|$ can be calculated using the equation in Theorem 5.5.1.

Results. We report the success rates of the attack simulations in Figure 5.8. The baseline success rate (first row) is the advantage $\mathsf{Adv}^{\mathsf{gs}}$, computed using the same attack strategy stated above except with no information about the bucket identifier. The following rows record the success rate of the attack for HPB and FSB with different parameter choices. The estimated security loss (Δ_q) can be calculated by subtracting the baseline success rate from the HPB and FSB attack success rates.

Protocol	Params	Bucket size		**Uncompromised**			
		Avg	max	$q = 1$	$q = 10$	$q = 10^2$	$q = 10^3$
Baseline	N/A	N/A	N/A	0.6	1.3	2.5	5.9
HPB	$l = 20^{\ddagger}$	244	303	33.7	49.7	63.0	71.0
	$l = 16^{\dagger}$	3,896	4,138	18.3	34.3	47.6	60.5
	$l = 12$	62,309	63,173	8.4	18.0	31.6	45.1
FSB	$\bar{q} = 1$	76	112	0.6	5.7	69.9	71.0
	$\bar{q} = 10$	908	1,010	0.6	1.3	5.5	70.0
	$\bar{q} = 10^2$	5,635	5,876	0.6	1.3	2.5	9.4
	$\bar{q} = 10^3$	21,107	21,550	0.6	1.3	2.5	5.8

Protocol	Params	Bucket size		**Compromised**			
		Avg	max	$q = 1$	$q = 10$	$q = 10^2$	$q = 10^3$
Baseline	N/A	N/A	N/A	37.4	50.4	51.4	52.6
HPB	$l = 20^{\ddagger}$	244	303	64.9	71.8	76.6	79.9
	$l = 16^{\dagger}$	3,896	4,138	58.2	65.0	71.1	75.7
	$l = 12$	62,309	63,173	53.5	58.2	63.9	70.0
FSB	$\bar{q} = 1$	76	112	51.3	53.3	79.4	79.9
	$\bar{q} = 10$	908	1,010	51.1	51.6	53.3	79.5
	$\bar{q} = 10^2$	5,635	5,876	50.7	51.5	52.1	54.7
	$\bar{q} = 10^3$	21,107	21,550	50.4	51.5	52.1	53.3

‡ HIBP uses $l = 20$ for its password-only C3 service.
† GPC uses $l = 16$ for username-password C3 service.

Figure 5.8: Comparison of attack success rate given q queries on different password-only C3 settings. All success rates are in percent (%) of the total number of samples (5,000). Bucket size, the number of passwords associated to a bucket, is measured on a random sample of 10,000 buckets.

The security loss from using HPB is devastating, especially for previously uncompromised users. Accessibility to the $l = 20$-bit hash prefix, used by HIBP [1], allows an attacker to compromise 34% of previously uncompromised users in just one guess. In fewer than 10^3 guesses, that attacker can compromise more than 70% of the accounts (12x more than the baseline success rate with 10^3 guesses). Google Password Checker (GPC) uses $l = 16$ for its username-password C3 service. Against GPC, an attacker only needs 10 guesses per account to compromise 34% of accounts. Reducing the prefix length l can decrease the attacker's advantage. However, that would also increase the bucket size. As we see for $l = 12$, the average

bucket size is 62,309, so the bandwidth required to perform the credential check would be high.

FSB resists guessing attacks much better than HPB does. For $q \leq \bar{q}$ the attacker gets no additional advantage, even with the estimated password distribution $\hat{p}_s$. The security loss for FSB when $q > \bar{q}$ is much smaller than that of HPB, even with smaller bucket sizes. For example, the additional advantage over the baseline against FSB with $q = 100$ and $\bar{q} = 10$ is only 3%, despite also having smaller bucket sizes than HPB with $l = 16$. Similarly for $\bar{q} = 100$, $\Delta_{10^3} = 3.6\%$. This is because the conditional distribution of passwords given an FSB bucket identifier is nearly uniform, making it harder for an attacker to guess the correct password in the bucket $\alpha(b)$ in q guesses.

For previously compromised users — users present in $\tilde{\mathcal{S}}$ — even the baseline success rate is very high: 37% of account passwords can be guessed in 1 guess and 53% can be guessed in fewer than 1,000 guesses. The advantage is supplemented even further with access to the hash prefix. As per the guessing strategy, the attacker first guesses the leaked passwords that are both associated to the user and in $\alpha(b)$. This turns out to be very effective. Due to the high baseline success rate the relative increase is low; nevertheless, in total, an attacker can guess the passwords of 80% of previously compromised users in fewer than 1,000 guesses. For FSB, the security loss for compromised users is comparable to the loss against uncompromised users for $q \leq \bar{q}$. Particularly for $\bar{q} = 10$ and $q = 100$, the attacker's additional success is only 1.9%. Similarly, for $\bar{q} = 100$ an attacker gets at most 2.1% additional advantage for a guessing budget of q=1,000. Interestingly, FSB performs significantly worse for compromised users compared to uncompromised users for $q = 1$. This is because the FSB bucketing strategy does not take into

account targeted password distributions, and the first guess in the compromised setting is based on the credential tweaking attack.

In our simulation, previously compromised users made up around 76% of the test set; it is unclear what is the actual proportion would be in the real world, so we do not combine results from the uncompromised and compromised settings.

As we can see, since the bucket sizes for FSB with $\bar{q} = 100$ and HPB with $l = 16$ are comparable, we will use $\bar{q} = 100$ as the parameter for FSB for further security and performance analysis. Note, GPC has a username-password C3 service and therefore, its bucket sizes will be larger. (See Figure 5.10.)

Password policy experiment. In the previous set of experiments, we assumed that the C3 server and the attacker use the same estimate of the password distribution. To simulate the effect when the attacker has a better estimate of the password distribution than the C3 server, we simulated a website which enforces a password policy. We assume that the policy is known to the attacker but not to the C3 server.

For our sample password policy, we required that passwords have at least 8 characters and that they must not be on Twitter's banned password list [102]. The test samples are drawn from T_{sp}, username-password pairs from T where passwords follow this policy, and the attacker is also given the ability to tailor their guesses to this policy. The server still stores all passwords in $\tilde{\mathcal{S}}$, without regard to this policy. Notably, the FSB scheme relies on a good estimate of the password distribution to be effective in distributing passwords evenly across buckets. Its estimate, when compared to the distribution of passwords in T_{sp}, should be less accurate than it was in the regular simulation, when compared to the password distribution from T.

Protocol	Uncompromised				Compromised			
	$q = 1$	10	10^2	10^3	$q = 1$	10	10^2	10^3
Baseline	0.1	0.4	1.2	3.1	37.9	46.7	47.0	47.8
HPB ($l = 16$)	9.6	17.8	27.0	41.2	51.0	55.7	59.5	63.8
FSB ($\bar{q} = 10^2$)	0.1	0.4	1.4	9.6	46.8	47.1	47.3	51.2

Figure 5.9: Attack success rate (in %) comparison for HPB with $l = 16$ (effectively GPC) and FSB with $\bar{q} = 10^2$ for password policy simulation. The first row records the baseline success rate $\mathsf{Adv}^{\mathsf{gs}}(q)$. There were 5,000 samples each from the uncompromised and compromised settings.

We chose the parameters $k = 16$ for HPB and $\bar{q} = 100$ for FSB, because they were the most representative of how the HPB and FSB bucketization schemes compare to each other. These parameters also lead to similar bucket sizes, with around 5,000 passwords per bucket. Overall, we see that the success rate of an attacker decreases in this simulation compared to the general experiment (without a password policy). This is likely due to the fact that after removing popular passwords, the remaining set of passwords that we can choose from has higher entropy, and each password is harder to guess. FSB still defends much better against the attack than HPB does, even though the password distribution estimate used by the FSB implementation is quite inaccurate, especially at the head of the distribution. FSB assigns larger probability estimates to passwords that are banned according to our password policy.

We also see that due to the inaccurate estimate by the C3 server for FSB, we start to see some security loss for an adversary with guessing budget $q = 100$. In the general simulation, the password estimate $\hat{p}_s$ used by the server was closer to p, so we didn't have any noticeable security loss where $q \leq \bar{q}$.

5.7 Performance Evaluation

In this section, we implement different approaches to checking compromised credentials and evaluate their computational overheads. For fair comparison, in addition to the algorithms we propose, FSB and IDB, we also implement HIBP and GPC with our breach dataset.

Setup. We build C3 services as serverless web applications that provide REST APIs. We used AWS Lambda [140] for the server-side computation and Amazon DynamoDB [141] to store the data. The benefit of using AWS Lambda is it can be easily deployed as Lambda@Edge and integrated with Amazon's content delivery network (CDN), called CloudFront [142]. (HIBP uses Cloudflare as CDN to serve more than 600,000 requests per day [125].) We used Javascript to implement the server and the client side functionalities. The server is implemented as a Node-JS app. We provisioned the Lambda workers to have maximum 3GB of memory. For cryptographic operations, we used a Node-JS library called Crypto [143].

For pre-processing and pre-computation of the data we used a desktop with an Intel Core i9 processor and 128 GB RAM. Though some of the computation (e.g., hash computations) can be expedited using GPUs, we did not use any for our experiment. We used the same machine to act as the client. The round trip network latency of the Lambda API from the client machine takes about 130 milliseconds. Recall that the breach dataset we use contains 255 million unique passwords and 749 million unique username-password pairs. (See Figure 5.7.)

To measure the performance of each scheme, we pick 20 random passwords from the test set T and run the full C3 protocol with each one. We report the average time taken for each run in Figure 5.10. In the figure, we also give the break down

of the time taken by the server and the client for different operations. The network latency had very high standard deviation (25%), though all other measurements had low ($< 1\%$) standard deviation compared to the mean.

HIBP. The implementation of HIBP is the simplest among the four schemes. The set of passwords in $\tilde{\mathcal{S}}$ is hashed using SHA256 and split into 2^{20} buckets based on the first 20 bits of the hash value (we picked SHA256 because we also used the same for FSB). Because the bucket sizes in HIBP are so small (< 500), each bucket is stored as a single value in a DynamoDB cell, where the key is the hash prefix. For larger leaked datasets, each bucket can be split into multiple cells. The client sends the 20 bit prefix of the SHA256 hash of their password, and the server responds with the corresponding bucket.

Among all the protocols HIBP is the fastest (but also weakest in terms of security). It takes only 208 ms on average to complete a query over WAN. Most of the time is spent in round-trip network latency and the query to DynamoDB. The only cryptographic operation on the client side is a SHA256 hash of the password, which takes less than 1 ms.

FSB. The implementation of FSB is more complicated than that of HIBP. Because we have more than 1 billion buckets for FSB and each password is replicated in potentially many buckets, storing all the buckets explicitly would require too much storage overhead. We use interval trees [139] to quickly recover the passwords in a bucket without explicitly storing each bucket. Each password w in the breach database is represented as an interval specified by $\beta_{\mathrm{FSB}}(w)$. We stored each node of the tree as a separate cell in DynamoDB. We retrieved the intervals (passwords) intersecting a particular value (bucket identifier) by querying the nodes stored in DynamoDB. FSB also needs an estimate of the password distribution to get the

interval range for a tree. We use $\hat{p}_s$ as described in Section 5.4. The description of $\hat{p}_s$ takes 8.9 MB of space that needs to be included as part of the client side code. This is only a one-time cost during client installation.

The depth of the interval tree is $\log N$, where N is the number of intervals (passwords) in the tree. Since each node in the tree is stored as a separate key-value pair in the database, one client query requires $\log N$ queries to DynamoDB. To reduce this cost, we split the interval tree into r trees over different ranges of intervals, such that the i-th tree is over the interval $[(i-1) \cdot \lfloor |\mathcal{B}|/r \rfloor\ ,\ i \cdot \lfloor |\mathcal{B}|/r \rfloor - 1]$. The passwords whose bucket intervals span across multiple ranges are present in all corresponding trees. We used $r = 64$, as it ensures each tree has around 4 million passwords, and the total storage overhead is less than 1% more than if we stored one large tree.

Each interval tree of 4 million passwords was generated in parallel and took 3 hours in our server. Each interval tree takes 400 MB of storage in DynamoDB, and in total 25 GB of space. FSB is the slowest among all the protocols, mainly due to multiple DynamoDB calls, which cumulatively take 273 ms (half of the total time, including network latency). This can be sped up by using a better implementation of interval trees on top of DynamoDB, such as storing a whole subtree in a DynamoDB cell instead of storing each tree node separately. We can also split the range of the range tree into more granular intervals to reduce each tree size. Nevertheless, as the round trip time for FSB is small (527 ms), we leave such optimization for future work. The maximum amount of memory used by the server is less than 81 MB during an API call.

On the client side, the computational overhead is minimal. The client performs one SHA256 hash computation. The network bandwidth consumed for sending the

Protocol	Client			Server		Total	Bucket
	Crypto	Server call	Comp	DB call	Crypto	time	size
HIBP	1	205	2	40	–	208	244
FSB	1	524	2	273	–	527	3,086
GPC	47	402	9	71	6	458	9,164
IDB	72	405	10	73	6	487	9,166

Figure 5.10: Time taken in milliseconds to make a C3 API call. The client and server columns contain the time taken to perform client side and server side operations respectively.

bucket of hash values from the server takes on average 261 KB.

IDB and GPC. Implementations of IDB and GPC are very similar. We used the same platform — AWS Lambda and DynamoDB — to implement these two schemes. All the hash computations used here are Argon2id with default parameters, since GPC in [2] uses Argon2. During precomputation, the server computes the Argon2 hash of each username-password pair and raises it to the power of the server's key K. These values can be further (fast) hashed to reduce their representation size, which saves disk space and bandwidth. However, hashing would make it difficult to rotate server key. We therefore store the exponentiated Argon2 hash values in the database, and hash them further during the online phase of the protocol. The hash values are indexed and bucketized based on either $\mathsf{H}^{(l)}(u\|w)$ (for GPC) or $\mathsf{H}^{(l)}(u)$ (for IDB). We used $l = 16$ for both GPC and IDB, as proposed in [2].

The server (for both IDB and GPC) only performs one elliptic curve exponentiation, which on average takes 6 ms. The remaining time incurred is from network latency and calling Amazon DynamoDB.

On the client side, one Argon2 hash has to be computed for GPC and two for IDB. Computing the Argon2 hash of the username-password pairs takes on

an average 20 ms on the desktop machine. We also tried the same Argon2 hash computation on a personal laptop (Macbook Pro), and it took 8 ms. In total, hashing and exponentiation takes 47 ms for GPC, and 72 ms (an additional 25 ms) for IDB. The cost of checking the bucket is also higher (compared to HIBP and FSB) due to larger bucket sizes.

IDB takes only 31 ms more time on average than GPC (due to one extra Argon2 hashing), while also leaking no additional information about the user's password. It is the most secure among all the protocols we discussed (should username-password pairs be available in the leak dataset), and runs in a reasonable time.

5.8 Deployment Discussion

Here we discuss different ways C3 services can be used and associated threats that need to be considered. A C3 service can be queried while creating a password — during registration or password change — to ensure the new password is not present in a leak. In this setting C3 is queried from a web server, and the client IP is potentially not revealed to the server. This, we believe, is a safer setting to use than the one we will discuss below.

In another scenario, a user can directly query a C3 service. A user can look for leaked passwords themselves by visiting a web site or using a browser plugin, such as 1Password [118] or Password Checkup [2]. This is the most prevalent use case of C3. For example, the client can regularly check with a C3 service to proactively safeguard user accounts from potential credential stuffing attacks.

However, there are several security concerns with this setting. Primarily, the

client's IP is revealed to the C3 server in this setting, making it easier for the attacker to deanonymize the user. Moreover, multiple queries from the same user can lead to a more devastating attack. Below we give two new threat models that need to considered for secure deployment of C3 services (where bucket identifiers depend on the password).

Regular password checks. A user or webservice might want to regularly check their passwords with C3 services. Therefore, a compromised C3 server may learn multiple queries from the same user, which can enable potentially powerful attacks. For FSB the bucket identifier is chosen randomly, so knowing multiple bucket identifiers for the same password will help an attacker narrow down the password search space and significantly improve attack success.

We can mitigate this problem for FSB by derandomizing the client side bucket selection using a client side state (e.g. browser cookie) so the client always selects the same bucket for the same password. We let the c be the client side cookie. To check a password w with the C3 server, the client picks the j^{th} bucket from the range $\beta(w)$, where $j \leftarrow f(w\|c) \mod |\beta(w)|$.

This derandomization ensures queries from the same device are deterministic (after the cookie is set). However, if the attacker can link queries of the same user from two different devices, the mitigation is ineffective. If the cookie is stolen from the client device, then the security of FSB is effectively reduced to that of HPB with similar bucket sizes.

Similarly, if an attacker can track the interaction history between a user and a C3 service, it can obtain better insight about the user's passwords. For example, if a user who regularly checks with a C3 service stops checking a particular bucket

identifier, that could mean the associated password may appear in the most up-to-date leaked dataset, and the attacker can use that information to guess the user's password(s).

Checking similar passwords. Another important issue is querying the C3 service with multiple correlated passwords. Some web services, like 1Password, use HIBP to check multiple passwords for a user. As shown by prior work, passwords chosen by the same user are often correlated [122, 121, 120]. An attacker who can see bucket identifiers of multiple correlated passwords can mount a stronger attack. Such an attack would require estimating the joint distribution over passwords. We leave analysis of this threat model for future work.

5.9 Related Work

There are existing cryptographic protocols that deal with threat models similar to ours. There has also been some previous work on distribution-sensitive cryptography for encrypted databases.

Private set intersection. Private set intersection (PSI) allows two parties to find the intersection between their private sets without revealing any additional information. Kiss et al. proposed an efficient PSI protocol for unequal set sizes based on oblivious pseudo-random functions (OPRF) [119], which performs well for sets with millions of elements. However, it is very expensive for larger set sizes (in the billions) and is therefore not feasible for checking leaked passwords. Other efficient solutions to PSI [128, 129, 130, 131] are also unable to handle datasets with billions of leaked passwords. The main overhead comes from the bandwidth

requirement — the server needs to send data proportional to the size of the leaked dataset.

Private information retrieval (PIR) [144] is another cryptographic primitive used to retrieve information from a server. Assuming the server's dataset is public, the client can use PIR to privately retrieve the entry corresponding to their password from the server. However, most advanced PIR schemes [145, 146] require exchanging large amounts of information over the network, so they are not useful for checking leaked passwords. PIR with two non-colluding servers can provide better security [147] than the bucketization-based C3 schemes, with communication complexity sub-polynomial in the size of the leaked dataset. However, deploying a C3 service with two non-colluding servers would be practically difficult.

Therefore, to circumvent these problems, the leaked dataset is divided into buckets before performing set intersection — a compromise between security and efficiency.

Compromised credential checking services. A number of servers are deployed in practice to provide C3 services, besides Google's Password Checkup [2] and HIBP [1]. For example, Vericlouds [148] and GhostProject [149] allow users to register with an email address, and regularly keeps the user aware of any leaked (sensitive) information associated with that email. Such services send information to the email address, and the user implicitly authenticates (prove ownership of the email) by having access to the email address. These services are not anonymous and must be used by the primary user. Moreover, these services cannot be used for password-only C3.

Distribution-sensitive cryptography. To build secure password-only C3 ser-

vice, we proposed FSB. Human-chosen passwords are non-uniformly distributed. FSB utilizes the knowledge of password distribution to provide better security. There have been a few works that similarly use knowledge of message distribution to design more secure protocols. Woodage et al. in [150] introduced secure sketches for password typos where sketches for popular passwords reveal less information than the sketches of unpopular passwords. Frequency-smoothing for deterministic encryption in encrypted databases replicates messages proportionally to their frequencies, and is shown to be effective against frequency attacks [151].

5.10 Conclusion

We explore different settings and threat models associated with checking compromised credentials (C3). The main concern is the secrecy of the user passwords that is being checked. We show, via simulations, that the existing industry deployed C3 services (such as HIBP and GPC) do not provide adequate security. Indeed an attacker who obtains the query to such a C3 service and the username of the querying user can severly damage the secrecy of the password. We give more secure C3 protocols for checking leaked passwords and username-password pairs. We implemented and deployed different C3 protocols on AWS Lambda and evaluated their computational and bandwidth overhead. We finish with several nuanced threat models and deployment discussions that should be considered when deploying C3 services.

CHAPTER 6

CONCLUSION

Password-based authentication (PBA) systems are by far the most widely deployed and used authentication mechanism. Passwords are a simple and intuitive way to authenticate users (if done in the correct way), and they do not require any additional hardware. Nevertheless, passwords are also blamed for being the weakest link in computer security. Repeated calls were made for their demise, but none of the proposed alternative techniques have so far been able to replace them entirely. Security and usability problems of passwords affect billions of users across the globe. And despite those problems, passwords will continue to be used for the foreseeable future. Therefore, improving the state of passwords and PBA systems is timely and necessary. In this dissertation, I showed how to do so using a novel approach that combines empirical techniques with analytical methods.

Usability and security of passwords are often thought to be fundamentally at odds. However, I explained in this dissertation how to rethink this connection and build next-generation PBA systems that provide better usability and better security. I used a novel technique, empiricism-informed secure system design, that combines analytical techniques with empirical tools to analyze and evaluate such systems, as well as to guide the process of designing and building.

I took advantage of the availability of large leaked password datasets to better understand human-chosen passwords. Knowledge of password distribution can be very useful in designing tailored solution for passwords. For example, NoCrack uses password models built for password cracking to generate decoys; TypTop uses knowledge of the password distribution to selectively deny typo correction on typos that would benefit an attacker; FSB replicates passwords based on their

probability to reduce leakage. I also showed how to instantiate those distribution-sensitive solutions using estimates of password distribution learned from leaked password datasets.

One of the core technical challenges in building distribution-sensitive solutions is that they must be agile to changes in the underlying distribution. A real-world password distribution might change over time. Also, due to different password policies, it might vary slightly across different websites. Ideally, it should be quick to re-parameterize a solution with a new password distribution and to understand its impact on the security of the system. Here comes the benefit of analytical bounds, which carefully distill out a clear and easily verifiable set of assumptions about the distribution, which if met, ensures that the security bounds automatically remain valid. Empirical methods then can be used to verify those assumptions based on the available data. This confluence of empiricism and analytical tools turned out to be very effective in designing better PBA systems, and can also be used for building other practical secure systems.

Despite the practicality and better security of such systems, one of the key open challenges is to get these systems deployed in practice. The adoption rate of new security techniques is very slow; it is even slower for user authentication systems. For example, despite the introduction of password hashing in the 1980s, several websites[1] can be found even today that do not hash their user passwords. Similarly, password managers, despite being a better solution, have very low adoption rate [152], because users are unsure about their effectiveness or do not trust their security. Site administrators might not want to turn on typo-tolerance (or even if they do, they want to be discrete about it) due to the fear of media backlash (e.g., [79, 77]): allowing small typos in passwords might seem insecure to some

[1] http://plaintextoffenders.com/

users. Lack of adoption of newer and better technologies is hurting both users and companies. We need to find better ways to communicate research results with the industry and the users.

Traditionally, users are disproportionately burdened with the security of passwords: users should pick strong and unique passwords for each of the accounts and change those passwords regularly. Often, prior policies require users to change their behavior (against their natural tendency) to improve security of passwords. Moving forward, users alone cannot be held responsible for improving security of passwords. We need to shift the burden onto PBA servers. By rethinking the tension between usability and security, PBA servers need to focus on how to help users inculcate and maintain better password habits. For example, allowing users to login with some typos might be a great way to encourage users to pick stronger and longer passwords. In addition to typo-tolerance, the websites should also ensure that these features are not exploited by attackers. By combining typo-tolerance with C3 services, the PBA server can defend against credential stuffing attacks while still allowing legitimate users to log in with small typos. Research should also look into how to defend typo-tolerant PBA systems from targeted password guessing attacks [122][2], such as credential tweaking attacks [120].

Finally, large scale PBA systems are typically deployed by industry, and are often protected, hiding the design details (via so-called security through obscurity). Most web services consider the authentication logs as sensitive data and do not share them with academic researchers. Others fear that revealing the details of their defense mechanism would advantage the attacker and reduce the effectiveness of their defense. As a result state-of-the art industrial PBA systems are not public,

[2]In targeted guessing attack, an attacker makes tailored guesses against a PBA systems based on auxiliary information about the user (such as the user's other leaked passwords or her personal information).

and academic research community might not be always working on problems that are at the bleeding edge of PBA systems. We need more collaboration between the academic community and industry to work on authentication related problems and build more usable and secure, next generation password-based authentication systems.

APPENDIX A

APPENDIX - CRACKING-RESISTANT PASSWORD VAULT

A.1 Constructing a PCFG

For completeness we describe our PCFG construction process, which uses techniques, with a few small embellishments, from prior work [39, 40]. We start by creating a parser for passwords based on some dictionaries and/or manual tuning. The parser itself uses a *base PCFG*. Using it, the passwords in a password leak are parsed into 'tokens' and the frequency of each token and each token combination is computed to generate a *trained PCFG*.

The process starts with input a password corpus (a list of passwords selected by real users in the wild) as well as a source dictionary. The dictionary is a list of words which we might expect to appear as or within passwords, with accompanying information about such words' expected likelyhood of occurrence. We construct a dictionary from the following two sources. The first, denoted **D1**, is the Wiktionary [153] list of the top 30,000 English words along with their frequency count as found in books of Project Gutenberg up through April 16, 2006. The second, denoted **D2**, is a list of Facebook first and last names and frequencies compiled by SkullSecurity [154]. After removing names with frequency less than five the final size of **D2** was 1,678,411. The Facebook first and last names capture not only names, surnames of persons, cities and popular entities but also a comprehensive list of popular words used by internet users. For example, **D2** covers modified spellings of words like 'gurl', acronyms like 'ciao' or 'afaik', etc.

Of course one could use other dictionaries for different languages, cultures, etc.,

but we leave the evaluation of such localization to future work.

The base weighted CFG. After manual inspection of the RockYou password corpus, we constructed an underlying grammar (CFG) for the base CFG of the following form:

```
S → W | D | Y | K | R | WW | WD | WY | WYD | WDY | WK | ...
W → <english-words>T | <names>T
D → <dates> | <numbers>
Y → <symbols>
K → <keyboard-sequence>
R → <repeated-characters>
T → Capitalize | ALL-CAPS | all-lower| L33t
```

where "..." signifies all remaining combinations of the non-terminals `W, D, Y, R` and `K`. The variables on the right hand side of each rule written within angle brackets (`<.*>`) are terminal place-holders. `<english-words>` and `<names>` are placeholders for sets of non-termanals associated to **D1** and **D2** respectively, while the rest of the placeholders are initialized with regular expressions. We explain the placeholders below in detail. The grammar is ambiguous (i.e., it can parse a password in multiple ways): 'password2015' could be parsed as {'pass', 'word', '2015'} or {'password', '2015'}. Obviously the latter seems like a more meaningful parsing. To disambiguate among different parse trees we assign some heuristic cost to each rule and we used the parse tree that incurs minimal cost. The costs can be seen as weighting various parsings, which is why we refer to the resulting CFG as weighted.

For the placeholder `<english-words>`, we specify a non-terminal rule for each word in **D1**. The cost of each rule for $X \in \mathbf{D1}$ is $1 - \frac{f_X}{\sum_X f_X}$, where f_X denotes

the frequency count of X in **D1**. For the placeholder <names>, the cost function is defined analogously, but using **D2**.

For the remaining non-terminals, we created simple regular expressions or rules that recognizes the underlying languages as closely as possible. Non-terminal <dates> produces dates in common formats (e.g. '08232013', '19790925', 2008 etc). We created a simple regex to parse dates; we found through hand tuning that we achieved the most reliable parsing by setting the cost of a date string str to be $\frac{|\text{str}|}{8}$ where |str| is the string's length. The constant 8 was chosen as the most popular date-based password length in practice and also the maximum length date that our regex will accept.

The non-terminal <keyboard-sequence> includes patterns of standard US keyboard sequences (e.g. 'qwerty', 'zxcvbn' etc). We achieved the most reliable parsing by setting the cost of a sequence str to be proportional to $\frac{c}{|\text{str}|}$ where c is the number of lateral changes in direction performed while typing 'str' on a standard US keyboard, a basic metric of typing complexity.

The non-terminal <repeated-characters> tries to predict whether a string is a sequence of identical characters, such as 'aaaaaaa','0000000'; we set the cost of this rule to $\left(\frac{\#\text{ unique characters in str}}{|\text{str}|}\right)^{|\text{str}|}$.

The non-terminal T leads to one of four special terminals that symbolize transforms applied in a post-processing step to the preceding string str generated by W. ALL-CAPS capitalizes all letters in the preceding string; Capitalize capitalizes the first letter; and all-lower sets all letters to lowercase. L33t applies a number of leetspeak transformations or selected capitalization. The cost for each of these rules is set to $\frac{c}{|\text{str}|}$, where c is 0 for all-lower, 1 for ALL-CAPS and

`Capitalizes`, and the number of transformed characters for `L33t`.

The resulting base weighted CFG permits the parsing of passwords in a password corpus, yielding insight into the structure of human-generated passwords. We consider for a given input the parse tree with the lowest cost under the heuristic explained above (i.e., sum of costs for rules in the parse tree). To identify the minimum cost parse tree we use a standard bottom-up dynamic programming approach similar to CYK [155]. Below, the term "parse tree" refers to the parse tree with minimum cost unless explicitly stated otherwise.

The trained PCFG. We now explain how we train a PCFG on the RockYou corpus, bootstrapping from the base weighted CFG.

We first fix some definitions used in this process. A production rule $l \to r$ can be specified as a pair (l, r). Every edge in a parse tree is a rule. A *rule set* is a lexicographically ordered set of rules with the same left-hand-side. A *rule list* is an ordered list of rules generated by depth-first search of the parse-tree of a string / password (with siblings taken in left to right order).

A CFG is completely specified as a set of rule sets. A PCFG is completely specified by an admissible assignment of probabilities to CFG rules. Let $\mathcal{S}$ be a rule set of size $|\mathcal{S}|$ and $p_{\mathcal{S}}(l \to r)$ be the probability of a rule $l \to r$ in $\mathcal{S}$. An admissible assignment of probabilities has the property that $\sum_{(l \to r) \in \mathcal{S}} p_{\mathcal{S}}(l \to r) = 1$. We refer to the probability distribution over rules in a rule set $\mathcal{S}$ for a given admissible assignment as its *induced* probability distribution.

To train a PCFG on our training corpus, we first parse each password using the base weighted CFG to obtain a parse tree for each password. The aggregate set $\mathcal{L} = \{R_1, \ldots, R_w\}$ of all rules in all these parse trees specifies a CFG. A rule

set $\mathcal{S}$ in this CFG is simply a set of all rules with a common left-hand side.

To specify a PCFG based on this CFG, we make use of the count of repetitions of each rule in $\mathcal{L}$ to construct an admissible assignment of probabilities. Let $f_{l\to r}$ denote the number of repetitions (the "frequency") of rule $l \to r$ in $\mathcal{L}$ and $f_{\mathcal{S}}$ denote the total number of repetitions of all rules in $\mathcal{S}$. Letting $p_{\mathcal{S}}(l \to r) = f_{l\to r}/f_{\mathcal{S}}$ yields an admissible assignment. We refer to this PCFG as the *trained PCFG*.

The non-terminals in the trained PCFG are as in the base PCFG, except that any special placeholder non-terminals that used post-processing (e.g., `L33t`) are at this stage replaced by a concrete (finite) set of non-terminals and/or terminals.

A.2 Securely Encoding Fractions

We have to define an encoding scheme for floating point numbers. We focus on empirically derived distributions, for which each probability that needs encoding is a fraction p/q where p is the frequency count and q is some cumulative total number of values observed. We have that $p \leq q$. In such cases, encoding can be done by choosing a large b-bit number subject to the constraint that it should be equal to p modulo q. For correctness, b has to be larger than number of bits in the binary representation of q. Encoding and decoding are defined in Figure A.1

Correctness is straightforward: by construction, $0 \leq r < 2^b$ and so

$$\begin{aligned} r \bmod q &= (x + p - (x \bmod q)) \bmod q \\ &= (x \bmod q) + (p \bmod q) - (x \bmod q) \\ &= p \bmod q \ . \end{aligned}$$

encode$_q(p)$**:**	**decode**$_q(r)$**:**
1. $x \leftarrow\!\!\$\ \{0,1\}^b$	1. $p \leftarrow r \bmod q$
2. $r \leftarrow x + p - (x \bmod q)$	2. **return** p
3. **if** $r \geq 2^b$ **then**	
4. $\quad r \leftarrow r - q$	
5. **return** r	

Figure A.1: encoding and decoding integer values.

In terms of security of encoding, we want the output values of the encoding scheme to be uniformly distributed over $\{0,1\}^b$ when p is uniformly selected from $[0, q-1]$. Otherwise an adversary can possibly rule out master passwords using the fact that encoded strings are distributed differently from uniform bits during decryption (under the wrong master password). Just for convenience we define $h = 2^b \bmod q$ and $l = \lfloor \frac{2^b}{q} \rfloor$. (Thus, $lq + h = 2^b$.) Now, consider four random variables A, B, C and D defined as follows: A and B are uniform distributions over $\{0,1\}^b$ and $\mathbb{Z}_q$, respectively; $C = A - (A \bmod q)$; and D is the output of the **encode** function for a randomly chosen $p \in [0, q-1]$. Note that D is almost equal to $C + B$, but for the potential additional modification made should $C + B$ has value equal to or larger than 2^b (see the conditional of line 3 of **encode**). Note also that C always takes values of the form kq for some $k \in \mathbb{Z}_{l+1}$. So, we have that the probability mass function for C is defined by:

$$\Pr[\, C = kq \,] = \begin{cases} \frac{q}{2^b} & \text{if } 0 \leq k < l \\ \\ \frac{h}{2^b} & k = l \end{cases}$$

Using this, we can see that for D it holds that:

$$\Pr[\, D = r \,] = \begin{cases} \frac{1}{2^b} & \text{if } r \in [0,\ (l-1)q + h) \\ \\ \frac{1}{2^b} + \frac{h}{2^b q} & \text{if } r \in [(l-1)q + h,\ lq) \\ \\ \frac{h}{2^b q} & \text{if } r \in [lq,\ 2^b) \end{cases}$$

The statistical distance between A and D, denoted by $\Delta(A, D)$, can then be calcu-

lated as follows.

$$\begin{aligned}\Delta(A, D) &= \sum_{x \in \{0,1\}^b} |\Pr[A = x] - \Pr[D = x]| \\ &= \frac{(q-h)h}{2^b q} + \frac{h(q-h)}{2^b q} \\ &\leq \frac{2(q-q/2)(q/2)}{2^b q} \leq \frac{q}{2^{b+1}}\end{aligned}$$

In a similar way we can show using a union bound that, while encoding m fractions, together the encoded values deviate from a uniform distribution over such values by at most $mq_{\max}/2^{b+1}$, where $q_{\max}$ is the largest q we shall need during encoding of fractions and $2^b \gg q_{\max}$. We found that we never encountered q more than 2^{32}, and do not need $m > 10^4$. Setting $b = 128$, and plugging in $q_{\max} = 2^{32}$ and $m = 10^4$ gives that the statistical distance from uniform for m encodings is strictly less than 2^{-83}, which suffices for our purposes.

APPENDIX B

APPENDIX - CORRECTING PASSWORD TYPOS

B.1 Secure Sketches

Secure sketches and fuzzy extractors, explored by Dodis et al. [83, 156], are designed to generate consistent, cryptographically strong keys from noisy secrets, such as biometric data. They may also be applied to passwords, as typographical errors in passwords can be modeled as noise. Dodis et al. proposed two ways to construct secure sketches for the edit-distance metric space; see Section 7 of [83]. They show how to use a low-distortion embedding for the edit-distance metric given by Ostrovsky and Rabani [157], and also describe a relaxed embedding for the edit-distance metric using c-shingles. The security losses for these constructions, as given in Proposition 7.2 and Theorem 7.5 of [83], are $t(\log F)2^{O\left(\sqrt{\log(n \log F) \log\log(n \log F)}\right)}$ and $\lceil \frac{n}{c} \rceil \log(n - c + 1) - (2c - 1)t\lceil \log(F^c + 1) \rceil$ respectively. Here, n is the size of the password, F is the alphabet size, t is the number of errors/edits tolerated, and c is a construction parameter denoting the size of the shingles. In our setting, typical values would be $n = 8$, $t = 1$, and $F = 96$. The value of c, according to Theorem 7.4, should be 1 in our setting (and the loss is an increasing function in c). Given these parameters, the entropy loss of the two secure sketches would be ≈ 91 bits and ≈ 31 bits respectively. The min-entropy of real world password distributions is only about ≤ 8 bits [45]. Thus known constructions provide no security guarantees in our context, and providing proven constructions that do would seem to require new techniques.

B.2 Sanitizing Caps-Lock Errors

As mentioned in the Chapter 3, preliminary analysis of the data revealed that a large fraction of errors was caused by accidental pressing of the caps-lock key. Measuring the rate of caps-lock errors is more challenging than for other typos, for two reasons. First, caps-lock key presses are not recordable via keystroke logging, and thus not directly detectable remotely. Second, if the user engages the caps-lock key while typing one password, there's a chance that it will remain on (erroneously) while the next one is entered. In MTurk, if an individual worker is to be permitted to enter more than one password—even across multiple HITs—propagation of caps-lock typos across passwords is therefore methodologically unavoidable. This second issue accounts for the (artificially) high rate of caps-lock typos observed in our experiments. We found that 76 HITs contributed to 1120 caps-lock errors.

To adjust for the effect of such propagation errors in determining the rate of caps-lock typos, we do the following. We define a caps-lock error as an incorrect password which, when the cases of all the letters are inverted, becomes correct. In sequentially processing the passwords in a HIT, we use a variable $\mathsf{CL\text{-}ERR} \in \{\mathtt{0}, \mathtt{1}\}$ to denote a heuristic determination as to whether the caps lock is in an error state when the user entered a password in a HIT. (An error state could either be that caps lock is on and the user should have typed lower-case letters, or caps lock is off and the user should have typed upper case letters.) We initially let $\mathsf{CL\text{-}ERR} = \mathtt{0}$. When we detect a caps-lock error in a password, we record it and set $\mathsf{CL\text{-}ERR} = \mathtt{1}$. If it is already the case that $\mathsf{CL\text{-}ERR} = \mathtt{1}$ when we reach a password in a HIT, we discard the password. That is, in such cases, we do not count it in our computation of error rates for any typo. Additionally, for every password in a HIT, we determine (heuristically) whether the caps lock has been turned off

during entry of the password. If the password contains at least one letter and the password was submitted correctly, then we set CL-ERR = 0.

In general, the intuition here is that we keep track heuristically of whether the caps-lock key appears to be engaged erroneously. If the entry of a password in a HIT has been affected by the state of the erroneously engaged caps-lock key, we treat it as "tainted," and thus discard it from our experiment. (We assume heuristically that caps-lock errors are independent of other typos. The global effect of discarding "tainted" passwords and not recording typos they contain in addition to caps-lock errors is small in any case.)

B.3 Complexity and Typo Likelihood

Our MTurk experiments revealed a significant initial finding regarding the frequency of typos. Typo rates in our study increased under the following three distinct metrics relating to password complexity.

Lexical diversity in passwords: One might suspect that more lexically diverse passwords—ones that include symbols, letters with different cases, numbers or some combination thereof—would be more prone to typos. We define four character classes: upper case letters, lower case letters, digits, and symbols. Now, based on how many of the four classes of characters are present within a password we can partition passwords into four buckets. For example, passwords containing characters from only one of the four classes are binned as bucket 1, passwords containing exactly two different classes of characters are bucket 2, etc. In our first sample of 100,000 passwords, there were very few lexically diverse passwords. RockYou has $< 0.2\%$ passwords with characters from all of the four character classes. So

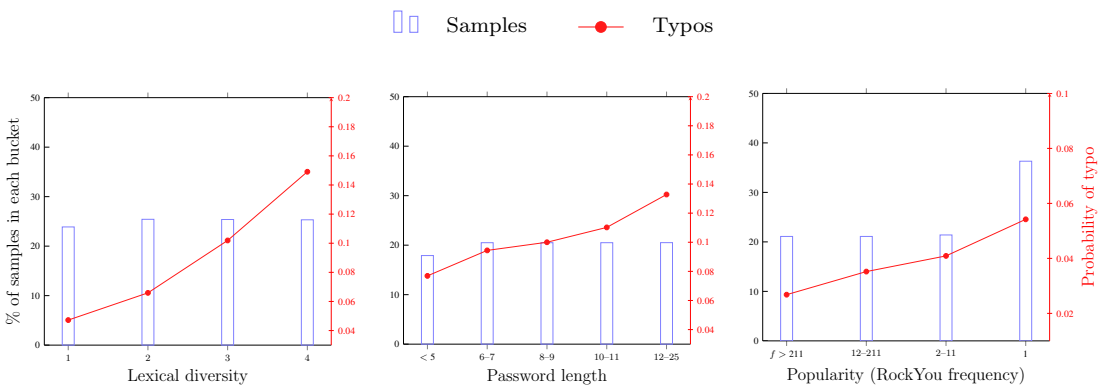


Figure B.1: Three experiments showing typo frequency relative to various partitions of passwords into buckets. Bucket size is indicated on the left of each figure, and corresponding typo rates on the right. **(Left)** Passwords are partitioned into four buckets based on diversity of character types. For each bucket we report the percentage of samples (blue bars) that fall in that bucket and what fraction of those samples are mistyped (red line). **(Middle)** Passwords are categorized into buckets by increasing order of length. **(Right)** Passwords are assigned to buckets by decreasing frequency (increasing unpopularity) in RockYou. Bucket frequency ranges are selected so that each bucket has roughly an equal number of samples.

we sample with replacement 5,000 passwords for each bucket from the empirical distribution of passwords in RockYou restricted to the passwords corresponding to the bucket. We performed the same typing experiment as described before but with new HITs created from these newly sampled passwords. In the left graph of Figure B.1, we present the percentage of passwords in each bucket that were mistyped.

Password length: We divide passwords into five groups based on their lengths, namely ≤ 5, 6–7, 8–9, 10–11, and ≥ 12. For each class, we compute the percentage of samples that lie in that class, along with the percentage of passwords in those samples that were mistyped. In the middle graph of Figure B.1 we show these numbers for each of the length groups. As one might expect, typo likelihood grows with password length.

Password popularity: We sort the list of sampled passwords for our MTurk

experiment based on their frequency counts in the RockYou leak. (Ties were broken alphabetically.) We then split the passwords into four buckets, adjusting their corresponding frequency ranges to ensure that buckets are of roughly equal size. (Some unevenness was unavoidable, as many passwords occur only once in the Rockyou leak.) For each bucket, we present the number of mistyped passwords in the right graph of Figure B.1. We can see the clear trend that passwords that are popular among RockYou users are more likely to be typed correctly. For example, passwords used by more than 211 users are 1.5 times more likely to mistyped than those used by only one user.

Discussion: password typing complexity. As noted above, lexical diversity, length, and popularity are related metrics. Inspection of the passwords within the various buckets used in the charts of Figure B.1 reveals that there is significant overlap between them. As one example, 18% of the passwords with lexical-diversity bucket 4 *both* have length ≥ 12 *and* are unpopular ($f = 1$).

The three metrics together highlight different aspects of the underlying and intuitive trend: some passwords are more difficult to type than others. It appears, moreover, that typos are more likely to surface in harder-to-guess passwords. Consequently, typo correction could help encourage users to adopt stronger passwords by easing the use of such passwords. We leave rigorous study of this hypothesis to future work, but note that it offers further potential motivation for our work.

B.4 Typist Speed and Typo Rate

As an enhancement of our experimental results in Section 4.6.1, we report on two experiments that provide further illumination of password features that lead

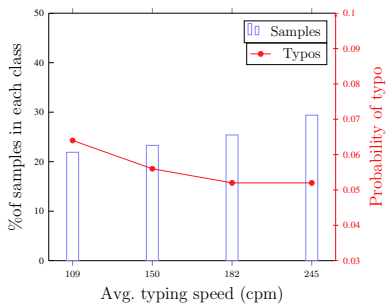


Figure B.2: The workers are divided into four quartiles based on their typing speed, and for each quartile we report the percentage of passwords that were mistyped.

to typos. These experiments further emphasize our observation that typo rates appear to increase with password complexity.

Typist and typo likelihood. In our MTurk experiments, we timestamped each character as it was typed during the experiments. We sorted the workers based on their average typing speed (characters-per-minute) and binned workers into four quartiles. For each quartile, we consider the subset of passwords that were typed by the typists whose speed falls in that quartile, and we compute the fraction of passwords that were mistyped in that subset. The data is reported in Figure B.2. We found that slow typists make more mistakes than faster typists. It could be that faster typists are also more skilled and so less likely to make mistakes.

Password entry time. We binned the passwords into four quartiles based on the time required to type those passwords. The fraction of typos in each of that quartile is reported in the middle chart of Figure B.3. Passwords that required more time on average to type are more likely to be mistyped.

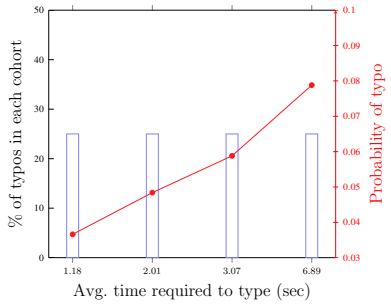


Figure B.3: Passwords are divided into four quartiles based on the amount of time spent by workers to type them, and then compute the fraction of passwords mistyped in each quartile.

B.5 Computing $\lambda_q^{\text{greedy}}$

We start by showing that computing the optimal q guesses to make against a relaxed checker is NP-hard in general. Later, we present an efficient approximate algorithm for the problem.

Definition 1 Best-q-guess. *Given a function $B : \mathcal{S} \to \mathcal{P}^*$, a password distribution p over $\mathcal{P} \subseteq \mathcal{S}$, and a query budget q, find a q-size subset $P \subseteq \mathcal{S}$, such that $\sum_{w \in C'} p(w)$ is maximized, where $C' = \bigcup_{\tilde{w} \in P} B(\tilde{w})$.*

Definition 2 Maximum coverage problem. *Given a ground set $E = \{e_1, e_2, \ldots e_n\}$, a collection of m subsets of E, $S = \{S_1, S_2, \ldots, S_m\}$, and a weight function $\gamma : E \to \mathbb{R}^+$ that assigns weights to each element $e \in E$, find a subset $C \subseteq S$ of size k, such that the following quantity is maximized, $\sum_{e \in C'} \gamma(e)$, where $C' = \bigcup_{S_i \in C} S_i$.*

The maximum coverage problem is known to be NP-hard [101]. We can thus prove the following theorem.

Theorem B.5.1 *If there is a polynomial time algorithm for best-q-guess, there exists a polynomial time algorithm for maximum coverage problem.*

Proof: We shall show a polynomial time reduction from the maximum coverage problem to the best-q-guess problem. To start with, we are given an instance of maximum coverage problem with (E, S, γ, k), and we want to construct an instance of best-q-guess problem. To do so, we set $\mathcal{P} = E$ and probability p as proportional to γ. (We might have to normalize γ to make it a probability distribution.) The function $B : \mathcal{S} \to \mathcal{P}^*$ is defined as follows. First add to $\mathcal{S}$ a set $W^* = \{\tilde{w}_i^*\}_{i=1}^m$, with $p(\tilde{w}_i^*) = 0$, and for each $S_i \in S$, set $B(\tilde{w}_i^*) = S_i \bigcup \{\tilde{w}_i^*\}$. For all other $\tilde{w} \in \mathcal{S} \setminus W^*$, let $B(\tilde{w}) = \{\tilde{w}\}$. This is a valid instance of the best-q-guess problem, and if we can find an polynomial time computable solution to this, we can solve the maximum weighted set cover problem in polynomial time. ▌

Nevertheless, a greedy algorithm can achieve $(1 - \frac{1}{e})$ times the optimal λ^{fuzzy} value and, as shown by Feige [101, 158], yields an optimal approximation for the maximum coverage problem. Naïve implementation of this greedy algorithm for best-q-guess, however, requires searching over exponentially many strings in $\mathcal{S}$. We can exploit the fact that in our setting, a small number of correctors are used, and these correctors are efficiently invertible. Additionally, password weights are highly non-uniform. Thus it is efficient to enumerate all balls above a certain threshold weight, yielding the efficient implementation of Feige's greedy approximation algorithm specified in Figure B.4. The algorithm intuition is this: as an invariant, in any iteration of the external while loop, all new balls (balls of $\tilde{w}$) pushed onto the heap have max-weight password w, and hence total weight $\leq b \cdot p(w)$. This observation enables a global selection of balls in weighted order.

nextPw()
Returns the passwords in $\mathcal{P}$ in decreasing order of their probability p.

FindGuesses(q)

```
/* B(w̃) = ball around w̃, and b = max_S |B(w̃)| */
/* N(w) = {w̃ | w ∈ B(w̃)} */
P ← 𝒫
A ← MaxHeap()     /* val(w̃) = p(B(w̃) ∩ P) */
g ← φ;
do {
  w ← nextPw()
  w̃_m ← A.popmax()
  while p(B(w̃_m) ∩ P) ≥ b · p(w) {
    g ← g ∪ {w̃_m}
    P ← P \ B(w̃_m)
    foreach w̃ ∈ {w̃ ∈ A | B(w̃) ∩ B(w̃_m) ∩ P ≠ φ}
      A.updateweight (w̃)
    w̃_m ← A.popmax()
  }
  A.heappush(w̃_m)
  foreach w̃ ∈ (N(w) \ A)
    A.heappush(w̃)
} while (|g| < q)
return g
```

Figure B.4: Figure presents a greedy algorithm to compute the best q guesses and thereby compute $\lambda_q^{\text{greedy}}$, for an attacker who estimates the the password distribution with p.

B.6 Proofs

We restate and then prove the free correction theorem from Section 3.5.4.

Theorem B.6.1 (Free Corrections Theorem) *Fix some password distribution p with support $\mathcal{P}$, a typo distribution τ, $0 < q < |\mathcal{P}|$ and an exact checker* ExChk. *Then for* OpChk *with any set of correctors $\mathcal{C}$, it holds that $\lambda_q^{\text{fuzzy}} = \lambda_q$.*

Proof: Let $\hat{S}$ be the optimal set of q strings which maximizes the total acceptance rate for the given checker OpChk. (Note that the order in which the queries are made does not change the success probability.) Let $B(S) = \cup_{\tilde{w} \in S} B(\tilde{w})$ for some

set S of strings in $\mathcal{S}$. Recall that $\lambda_q = \sum_{i=1}^{q} p(w_i)$ is the sum of the probabilities of q most probable passwords in $\mathcal{P}$. On the other hand, $\lambda_q^{\text{fuzzy}} = p\left(B(\hat{S})\right)$. The above holds because $B(\tilde{w})$ is the set of passwords checked by OpChk for a given string $\tilde{w}$.

The checker OpChk ensures that the cumulative probability mass of any ball is less than or equal to $p(w_q)$ whenever the size of the ball is more than 1, but, if the size is one, the cumulative probability can be more than $p(w_q)$. So, we split $\hat{S}$ into two distinct groups $\hat{S}_1$ and $\hat{S}_{>1}$, where $\hat{S}_1$ is the set of all strings in S whose ball sizes are exactly one, and $\hat{S}_{>1} = \hat{S} \setminus \hat{S}_1$. We can claim following two inequalities.

$$p\left(B(\hat{S}_1)\right) \leq \sum_{i=1}^{|\hat{S}_1|} p(w_i) \tag{B.1}$$

$$p\left(B(\hat{S}_{>1})\right) \leq |\hat{S}_{<1}| p(w_q) \leq \sum_{i=|\hat{S}_1|+1}^{q} p(w_i) \tag{B.2}$$

Equation (B.1) is true because the $|B(\hat{S}_1)| = |\hat{S}_1|$, and the right hand side is the highest cumulative probability that any set of that size can achieve under p. Equation (B.2) is true because of the facts that $p(B(\tilde{w})) \leq p(w_q)$ for all $\tilde{w} \in \hat{S}_{>1}$, and $p(w_i) \geq p(w_q)$ for all $i \geq q$. So, by a union bound over $B(\hat{S}_{>1})$, we can achieve that inequality. We can add the two inequalities to obtain our desired result.

$$p\left(B(\hat{S})\right) = p\left(B(\hat{S}_1)\right) + p\left(B(\hat{S}_{>1})\right) \leq \sum_{i=1}^{q} p(w_i)$$

To show strict equality, simply observe that an attacker against OpChk can always choose the q most probable passwords to guess and achieve a success rate of λ_q. Thus $\lambda_q^{\text{fuzzy}} = \lambda_q$. ∎

Theorem B.6.2 *Fix $q > 0$, a distribution pair (p, τ), and a corrector set $\mathcal{C}$. Define* OpChk *to work over $\mathcal{C}$ and let* Chk *work for a set of correctors $\mathcal{C}' \subseteq \mathcal{C}$. If $\Delta_q(\mathsf{Chk}) = 0$, then* $\mathsf{Utility}(\mathsf{Chk}) \leq \mathsf{Utility}(\mathsf{OpChk})$.

Proof: First recollect utility of any checker Chk is defined as

$$\begin{aligned}\mathsf{Utility(Chk)} &= \Pr\left[\,\mathsf{ACC(Chk)} \Rightarrow \mathsf{true}\,\right] \\ &= \sum_{\tilde{w}\in\mathcal{S}} \sum_{w\in B(\tilde{w})} p(w)\cdot\tau_{\tilde{w}}(w),\end{aligned}$$

where $B(\tilde{w})$ is the ball of $\tilde{w}$ under Chk.

Let assume for contradiction that there exists a checker Chk which uses only the correctors in $\mathcal{C}$, achieves a $\Delta_q(\mathsf{Chk}) = 0$ and still beats the OpChk in utility, that is, $\mathsf{Utility(Chk)} > \mathsf{Utility(OpChk)}$. Let denote a ball of OpChk by $B(\cdot)$ and a ball of Chk by $\tilde{B}(\cdot)$. So, if $\mathsf{Utility(Chk)} > \mathsf{Utility(OpChk)}$, then there exists at least one $\tilde{w} \in \mathcal{S}$ such that

$$\sum_{w\in\tilde{B}(\tilde{w})} p(w)\cdot\tau_w(\tilde{w}) > \sum_{w\in B(\tilde{w})} p(w)\cdot\tau_w(\tilde{w}). \tag{B.3}$$

Now, by construction, the optimal checker OpChk selects the $B(\tilde{w})$ that maximizes the utility under the constraint that no ball of size 1 has higher cumulative mass than $p(w_q)$. Here by utility we mean the sum $\sum_{w\in B(w)} p(w)\cdot\tau_w(\tilde{w})$. (See Eqn. 3.5.4.) The checker Chk can achieve higher utility only if it violates one of the two constraints in (3.5.4). The first constraint, required for completeness, is inviolable. The second constraint determines security; if $p(\tilde{B}(\tilde{w})) > p(w_q)$ when $|\tilde{B}(\tilde{w})| > 1$, then the security loss $\Delta_q(\mathsf{Chk}) > 0$ according to Lemma B.6.1. Thus there cannot exist any $\tilde{w} \in \mathcal{S}$ fulfilling Eqn. B.3. Thus the assumption $\mathsf{Utility(Chk)} > \mathsf{Utility(OpChk)}$ is false. ∎

Lemma B.6.1 *For any password and typo distribution pair (p, τ), checker Chk, and parameter $0 < q < |\mathcal{P}|$, if there exists a string $\tilde{w} \in \mathcal{S}$, s.t. $|B(\tilde{w})| > 1$ and $p(B(\tilde{w})) > p(w_q)$, then $\Delta_q > 0$.*

Proof: Security loss $\Delta_q > 0$ implies that $\lambda_q^{\text{fuzzy}} > \lambda_q$. Let $\mathcal{P}_q = \{w_1, \ldots, w_q\}$ and

recall that $\lambda_q = p(\mathcal{P}_q)$. Recall too that:

$$\lambda_q^{\text{fuzzy}} = \max_{S \subseteq \mathcal{S}} p(B(S)), \quad \text{where } |S| = q.$$

First set $S \leftarrow (\mathcal{P}_q \setminus B(\tilde{w})) \cup \{\tilde{w}\}$. Clearly $\lambda_q^{\text{fuzzy}} \geq p(B(S))$. If we look at the union of balls of the strings in the set S,

$$B(S) \supseteq \mathcal{P}_q \cup B(\tilde{w})$$

$$\Rightarrow p(B(S)) \geq p(\mathcal{P}_q) + p(B(\tilde{w}) \setminus \mathcal{P}_q)$$

Now, if $B(\tilde{w}) \setminus \mathcal{P}_q \neq \phi$, then clearly $\lambda_q^{\text{fuzzy}} \geq p(B(S)) > p(\mathcal{P}_q)$, and so $\Delta_q > 0$.

If $B(\tilde{w}) \setminus \mathcal{P}_q = \phi$, then $|S| < q$, as $|B(\tilde{w})| > 1$. Thus as long as there exists a password $w' \in \mathcal{P} \setminus S$ such that $p(w') > 0$, we can add w' to S, resulting in $p(B(S)) > p(\mathcal{P}_q)$. This concludes the proof. ∎

B.7 Toy Example of Poor Ball Estimation

Consider the following toy example of the attacker's estimated distribution $\hat{p}$ and the actual challenge distribution p:

Attacker's estimate	
w	$\hat{p}(w)$
`123456`	1/3
`password`	1/4
`Password`	1/4
`qwerty`	1/6

Actual distribution	
w	$p(w)$
`123456`	1/2
`password`	1/5
`Password`	1/5
`asdffghj`	1/10

The best guess of the attacker against ExChk is `123456`, which yields success rate 1/2. If the attacker wants to optimize her guess in the presence of a typo tolerant checker, e.g., Chk-All with correctors $\mathcal{C}_{\text{top2}}$, the she select as her first guess is `password` (in whose ball `Password` lies), yielding success probability only 2/5.

APPENDIX C

APPENDIX - PERSONALIZED PASSWORD TYPO CORRECTION

C.1 Benefits of the PLFU Caching Scheme

We now describe the intuition behind the utility benefits of the probabilistic least frequently used (PLFU) caching scheme over its deterministic counterpart (LFU). Recall that during each PLFU cache update, the cached typo $\tilde{w}_o$ is replaced with the wait listed typo $\tilde{w}_n$ with probability $\nu = f_{\tilde{w}_n}/(f_{\tilde{w}_n} + f_{\tilde{w}_o})$, where f denotes the frequency count of the typo in the subscript. The main goal of the PLFU caching scheme is to let the cache be agile enough to adapt and change, while simultaneously increasing the likelihood that the most useful typos ultimately stay in the cache.

The PLFU scheme has two key benefits over the non-probabilistic LFU scheme. First, it makes it more likely that a typo which is repeated in small amounts over many login attempts (and thus is likely to increase usability if cached) ultimately enters the cache, as opposed to one which appears multiple times in a single (and possibly anomalous) login attempt. For example, if a typo $\tilde{w}_n$ is repeated once across r login attempts in which the least frequently used typo $\tilde{w}_o$ is not entered at all, the probability that $\tilde{w}_n$ will not enter the cache is approximately $(1 - 1/(1 + f_{\tilde{w}_o}))^r$, which is less than $1/e$ if $r \approx f_{\tilde{w}_o} > 2$. Conversely, if the same typo appears $r \approx f_{\tilde{w}_o}$ times in a single login attempt, the probability that it does not then enter the cache is $(1 - r/(f_{\tilde{w}_o} + r)) \approx 1/2 > 1/e$.

Second, by setting the frequency count for the newly cached typo to $f_{\tilde{w}_n} + f_{\tilde{w}_o}$, the PLFU scheme increases the eventual stability of the cache, as the increasing

frequency counts of the cached typos mean that the probability that they are replaced by a wait listed typo decreases over time. In contrast the LFU caching scheme updates the cache on each login attempt, increasing the chance that useful typos are accidentally evicted from the cache.

C.2 Modeling the Typo Distribution τ_w

In this section, we describe the procedure with which we built the typo model τ used for the simulations on Page 143. We use a supervised training method to learn the typo distribution using the typo data collected in our MTurk study (Section 4.6.1) and the data released with [30]. Our approach is inspired by that of Houser et al. [159].

A simple way to build the typo model would be to compute the frequency distribution of typos for each password. However, our data set contains only $30,000$ typos of $20,000$ distinct passwords in total. As such, both the set of passwords on which we have typo data, and the amount of typo data we have for the individual passwords, is too small to build a good frequency based model. Therefore, we make two simplifying assumptions about typographical errors: that a typo of a given character in a password is influenced by the characters very close to it, and that this typo is independent of the characters in the remainder of the string.

Given a list of pairs of passwords and typos, we first align each pair by inserting a special symbol "␣" zero or more times (as required), such that the resulting pair of strings are of the same length, and have the same DL distance as the originals. For example, the password-typo pair (`password`, `pasword`) may be aligned to (`password`, `pa␣sword`). If there are multiple alignments possible, we consider

all of them. For each of the aligned password-typo pairs $(w, \tilde{w})$, we take the set of substring pairs $(w_{i:j}, \tilde{w}_{i:j})$ for all $0 \leq j \leq k;\ 0 \leq i \leq |w| - j$, and compute the frequency distribution of those pairs across all the aligned password-typo pairs. Here $|x|$ denotes the length of the string x; $x_{i:n}$ denotes the sub-string of length n of x beginning at location $i \leq |x| - n$; and k is a parameter of the model. We let E denote the frequency distribution of these pairs of strings, and will use it in subsequent steps to compute the typo probability of a given password.

We define an edit as a triplet (i, l, r), where i denotes a location in the string, and l and r are strings of length at most k. An edit (i, l, r) is valid for a password w if $w_{i:|l|} = l$. Transforming a password w by applying a valid edit (i, l, r) means, replacing $w_{i:|l|}$ with the sub-string r. For a given password w we let E_w denote the set of all possible valid edits in the set $\mathbb{Z}_{|w|} \times E$. We assign weights to each edit (i, l, r) in E_w as the frequency of the pair (l, r) according to the frequency distribution E, divided by the number of locations $j \in [0, |w|]$ for which (j, l, r) is a valid edit for w. The weights are then normalized to define a probability distribution P_w over the valid edits of w.

With this in place, the process of sampling a typo of a password w according to the typo model τ_w is reduced to sampling an edit from P_w and applying it to w. Note that there could be multiple edits of a password w which result in the same typo $\tilde{w}$. Thus, $\tau_w(\tilde{w})$ is equal to the sum of the probabilities of all edits that transform w into $\tilde{w}$.

Efficacy of the typo model. We use the average log likelihood — which is a typical measure for evaluating generative models — to gauge the efficacy of our typo model. We compare our model against the naive uniform model, that assigns uniform probabilities to all the typos. Note, a model is better if the average log

likelihood value is higher.

We perform a cross validation of our model over five 80:20 train-test splits of the data set of password / typo pairs. For each split, we train our typo model using the training data, and compute the average log likelihood of test samples as the average of $\log \tau_w(\tilde{w})$ taken over all pairs $(w, \tilde{w})$ in the test data. We find that the average log likelihood of the test data according to our model is –7.2 with standard deviation 0.4, which is much better than base uniform model, which obtains an average likelihood of –11. This suggests that our model captures the real world typo distribution fairly well.

C.3 Proofs from Section 4.5

Proof of Lemma 4.5.1. We begin by proving Lemma 4.5.1 which we shall utilize in subsequent analysis.

Proof: We argue by a series of game hops. Let game G_0 denote game $\mathrm{OFFDIST}^{\mathcal{A}}_{\Pi,\mathcal{T}}$ (as defined in Figure 4.4), so

$$\mathsf{Adv}^{\mathrm{offdist}}_{\Pi,\mathcal{T}}(\mathcal{A}) = 2 \cdot \left| \Pr[\, \mathrm{G}_0 \Rightarrow 1 \,] - \frac{1}{2} \right| .$$

Recall that the cache of TypTop stores up to $(t+1)$ ciphertexts, each of which corresponds to a password-based encryption (using the canonical PBE scheme $\mathsf{PBE}[\mathsf{SH}, \mathsf{SE}]$) of the secret key of the PKE scheme sk under the real password and each of the (at most t) cached typos.

We define a new game G_1 which is identical to G_0 except that the keys sampled to encrypt cached ciphertexts are now sampled without replacement. In more detail, when a new typo is cached during the evolution of the challenge state

s_n, a salt is chosen $\mathsf{sa} \leftarrow\!\!\$\ \{0,1\}^{\ell_{\mathsf{salt}}}$, and the cached ciphertext is computed as $c \leftarrow\!\!\$\ \mathsf{E}(\mathsf{SH}(\mathsf{sa}||\tilde{w}), sk)$. In game G_0 oracle SH responds to fresh queries of this form by returning $\mathsf{k}_j \leftarrow\!\!\$\ \{0,1\}^{\kappa}$, whereas in G_1 oracle SH samples these keys without replacement. In the distinguishing phase of G_1, SH returns to sampling keys with replacement. These games run identically unless two keys sampled during the computation of these cached ciphertexts collide. Since at most $(t \cdot (n+1)+1)$ such keys are sampled while processing a transcript of length n (where the $(t+1)$ term corresponds to the maximum number of keys sampled to encrypt the ciphertexts in the initial cache, and the $t \cdot n$ term arises as processing each of the n typos in the transcript can introduce at most t new ciphertexts in the typo cache), it follows that

$$2 \cdot |\Pr[\,\mathrm{G}_0 \Rightarrow 1\,] - \Pr[\,\mathrm{G}_1 \Rightarrow 1\,]| \leq \frac{(t \cdot (n+1)+1)^2}{2^{\kappa}}\ .$$

Next we define game G_2 which is identical to G_1 except we replace $\mathsf{Checker}[\Pi]$ with $\mathsf{PChecker}[\Pi]$ and a sequence of statements that encrypt the final typo cache, state and wait list returned by it as specified by the scheme. Notice that these games run identically unless during the process of updating the state we find two distinct keys $\mathsf{k}_1 \neq \mathsf{k}_2$ such that $\mathsf{D}_{\mathsf{k}_2}(\mathsf{E}_{\mathsf{k}_1}(sk)) \neq \perp$ where sk denotes the secret key of the PKE scheme which is encrypted under each of the cached typos. As such the fundamental lemma of game playing [160] implies that the gap between game G_1 and G_2 is upper-bounded by the probability that this event occurs. Notice that we can further upper bound this probability by $\mathsf{Adv}^{\mathrm{rob}}_{\mathsf{SE}}(\mathcal{R})$ as follows. Consider an adversary $\mathcal{R}$ in game $\mathrm{ROB}^{\mathcal{R}}_{\mathsf{SE}}$ who simply executes the game G_1, simulating SH by sampling random strings without replacement, and checking if there ever exists a typo cache ciphertext $\mathsf{E}_{\mathsf{k}_1}(sk)$ that decrypts under some subsequently sampled $\mathsf{k}_2 \neq \mathsf{k}_1$ (recall that since G_1 samples without replacement, all keys are distinct).

The gap between these two games is upper bounded by the probability that this event occurs, and so the robustness of the encryption scheme implies that

$$2 \cdot |\Pr[\,\mathrm{G}_1 \Rightarrow 1\,] - \Pr[\,\mathrm{G}_2 \Rightarrow 1\,]| \leq 2 \cdot \mathsf{Adv}^{\mathrm{rob}}_{\mathsf{SE}}(\mathcal{R}) \,.$$

Next we define game G_3 which is identical to G_2 except we return SH to sampling keys with replacement. An analogous argument to that above bounding the probability that two keys collide ensures that the gap between these games is again bounded above by $\frac{(t\cdot(n+1)+1)^2}{2^\kappa}$.

Notice that G_3 is identical to game $\overline{\mathrm{OFFDIST}}_{\Pi,\mathcal{T}}$, and so may be perfectly simulated by an attacker $\mathcal{A}'$ in this game. On input challenge state s_n, attacker $\mathcal{A}'$ passes this state to $\mathcal{A}$ in game G_3, simulating $\mathcal{A}$'s random oracle by querying his own oracle SH, and returning the responses. Since $\mathcal{A}'$ makes precisely the same set of oracle queries as $\mathcal{A}$, it follows that if $\mathcal{A}$ makes at most q queries, then $\mathcal{A}'$ does also. At the end of the game $\mathcal{A}'$ outputs whatever bit $\mathcal{A}$ does, and so

$$2 \cdot \left|\Pr[\,\mathrm{G}_3 \Rightarrow 1\,] - \frac{1}{2}\right| = \mathsf{Adv}^{\overline{\mathrm{offdist}}}_{\Pi,\mathcal{T}}(\mathcal{A}') \,,$$

concluding the proof. ∎

Proof of Theorem 4.5.1. We now provide the full proof of Theorem 4.5.1.

Proof: Consider an adversary $\mathcal{A}$ in game $\mathrm{OFFDIST}^{\mathcal{A}}_{\Pi,\mathcal{T}}$. Recall that by Lemma 4.5.1, there exist adversaries $\mathcal{A}'$ and $\mathcal{R}$ both running in time approximately that of $\mathcal{A}$ and where $\mathcal{A}'$ makes the same number of oracle queries as $\mathcal{A}$ such that,

$$\mathsf{Adv}^{\mathrm{offdist}}_{\Pi,\mathcal{T}}(\mathcal{A}) \leq \mathsf{Adv}^{\overline{\mathrm{offdist}}}_{\Pi,\mathcal{T}}(\mathcal{A}') + 2 \cdot \mathsf{Adv}^{\mathrm{rob}}_{\mathsf{SE}}(\mathcal{R}) + \frac{(t \cdot (n+1)+1)^2}{2^{(\kappa-1)}} \,;$$

so it is sufficient to upper bound the success probability of an attacker $\mathcal{A}'$ in game

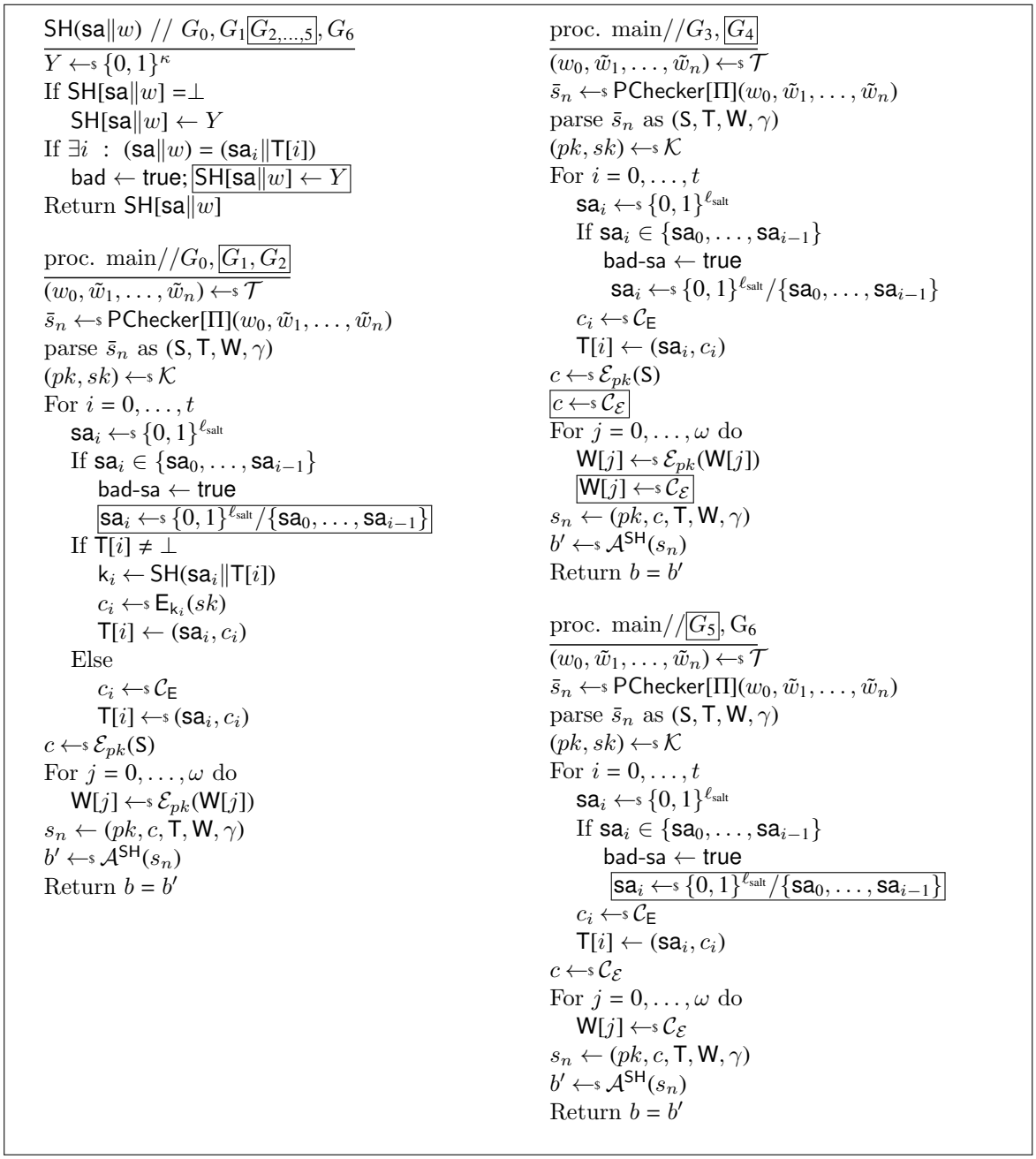


Figure C.1: Games used in the proof of Theorem 4.5.1.

$\overline{\text{OFFDIST}}^{\mathcal{A}'}_{\Pi,\mathcal{T}}$. We argue by a series of game hops, shown in Figure C.1. We begin by defining game G_0, which is identical to game $\overline{\text{OFFDIST}}^{\mathcal{A}'}_{\Pi,\mathcal{T}}$ with $b = 0$. We also set two flags, bad-sa and bad neither of which affect the outcome of the game.

Next we define game G_1 which is identical to G_0 except the salts used by the

canonical PBE scheme to compute the ciphertexts in the challenge state are now sampled without replacement. Games G_0 and G_1 run identically unless the flag bad-sa is set to true. Since there are at most $t+1$ salts sampled, the birthday bound and the fundamental lemma of game playing therefore imply that this transition is bounded above by $\frac{(t+1)^2}{2^{\ell_{\text{salt}}+1}}$, and so

$$\begin{aligned}|\Pr[\,G_0 \Rightarrow 1\,] - \Pr[\,G_0 \Rightarrow 1\,]| &\leq \Pr[\,\mathsf{bad\text{-}sa} = \mathsf{true} \text{ in game } G_1\,] \\ &\leq \frac{(t+1)^2}{2^{\ell_{\text{salt}}+1}}\ .\end{aligned}$$

We now define game G_2 which is identical to G_1 except we change the way in which the random oracle SH responds to queries. Now if in the guessing phase $\mathcal{A}$ queries SH on a salt / password pair (sa, w) on which it was queried during the computation of the challenge state s_n, it responds with an independent random string $\mathsf{k} \leftarrow\!\!\$\ \{0,1\}^\kappa$ updating its hash table to this new value, as opposed to the previously used value. Accordingly in G_2 the keys k used by the SE encryption scheme are now random and independent of the underlying password. Games G_1 and G_2 run identically unless $\mathcal{A}$ manages to guess and query SH on one of the cached passwords and corresponding salt; an event we mark by setting a flag bad = true. The fundamental lemma of game playing then implies that,

$$|\Pr[\,G_1 \Rightarrow 1\,] - \Pr[\,G_2 \Rightarrow 1\,]| \leq \Pr[\,\mathsf{bad} = \mathsf{true} \text{ in game } G_2\,]\ ,$$

a probability which we shall upper bound in a later game.

Next we define game G_3, which is identical to G_2 except we replace all symmetric encryptions with random ciphertexts. This transition is bounded by a reduction to the MKROR security of SE. Formally, let $\mathcal{B}_1$ be an adversary in game $\text{MKROR}^{\mathcal{B}_1,t}_{\mathsf{SE}}$ with challenge bit b'. Adversary $\mathcal{B}_1$ runs $(w_0, \tilde{w}_1, \ldots, \tilde{w}_n) \leftarrow\!\!\$\ \mathcal{T}$, followed by $\bar{s}_n \leftarrow\!\!\$\ \mathsf{PChecker}[\Pi](w_0, \tilde{w}_1, \ldots, \tilde{w}_n)$, and generates a public / secret key pair $(pk, sk) \leftarrow\!\!\$\ \mathcal{K}$. $\mathcal{B}_1$ then constructs the encrypted state s_n as follows. $\mathcal{B}_1$ first

chooses $\mathsf{sa}_i \leftarrow\!\!\$\ \{0,1\}^\kappa$ and uses his RoR oracle to compute $c_i = \mathsf{RoR}(i, sk)$ for $i = 0, \ldots, t$, placing the salt / ciphertext pairs in the appropriate positions in the cache. (Recall that from game G_2, the symmetric keys used to create the ciphertexts in s_n are random and independent of the salts and underlying passwords, and so identically distributed to those used by $\mathcal{B}_1$'s RoR oracle). Next, $\mathcal{B}_1$ encrypts S and the entries in W under the public key pk, and assembles challenge state s_n accordingly. Finally $\mathcal{B}_1$ passes s_n to $\mathcal{A}'$, simulating queries to SH by returning a random bit string to each fresh query, and at the end of the game outputs whatever bit $\mathcal{A}'$ does. Notice that if $b' = 1$ and $\mathcal{B}_1$ is receiving real encryptions from the RoR oracle then this perfectly simulates G_2, and if $b' = 1$ this perfectly simulates G_3. It follows that,

$$
\begin{aligned}
&|\Pr[\,\mathrm{G}_2 \Rightarrow 1\,] - \Pr[\,\mathrm{G}_3 \Rightarrow 1\,]| \\
&\qquad = |\Pr[\,\mathcal{B}_1 \Rightarrow 1 \mid b' = 0\,] - \Pr[\,\mathcal{B}_1 \Rightarrow 1 \mid b' = 1\,]| \\
&\qquad \leq \mathsf{Adv}^{\mathrm{mkror}}_{\mathsf{SE}}(\mathcal{B}_1, t).
\end{aligned}
$$

We can similarly show that the probability that bad is set in G_3 is close to the probability that it is set in G_2 via a separate reduction to the MKROR security of SE. Formally let $\mathcal{B}_2$ be an adversary in game $\mathrm{MKROR}^{\mathcal{B}_2}_{\mathsf{SE},t}$. $\mathcal{B}_2$ constructs the simulated state s_n as described above, using its RoR oracle to compute the symmetric encryptions in the state. However now when $\mathcal{B}_2$ passes s_n to $\mathcal{A}'$, it watches the queries $\mathcal{A}'$ makes to SH. If there exists a query $(\mathsf{sa}_i||\tilde{w}_i)$ where $i \in [0, t]$ such that $\tilde{w}_i$ is equal to the typo corresponding to position i in the cache (in which case the flag $\mathsf{bad} \leftarrow \mathsf{true}$), $\mathcal{B}_2$ outputs 1; else it returns 0. By the same argument made

above, it follows that,

$$\begin{aligned}\Pr[\,\mathsf{bad} = \mathsf{true} \text{ in } \mathrm{G}_2\,] &\leq \Pr[\,\mathsf{bad} = \mathsf{true} \text{ in } \mathrm{G}_3\,] \\ &\quad + |\Pr[\,\mathcal{B}_2 \Rightarrow 1 \mid b' = 0\,] - \Pr[\,\mathcal{B}_2 \Rightarrow 1 \mid b' = 1\,]| \\ &\leq \Pr[\,\mathsf{bad} = \mathsf{true} \text{ in } \mathrm{G}_3\,] + \mathsf{Adv}^{\mathrm{mkror}}_{\mathsf{SE}}(\mathcal{B}_2, t)\ .\end{aligned}$$

We may now define a third adversary $\mathcal{B}$ in game $\mathrm{MKROR}^{\mathcal{B},t}_{\mathsf{SE}}$ who flips a bit and depending on the outcome runs either $\mathcal{B}_1$ or $\mathcal{B}_2$ and then outputs the same bit as that adversary. It is easy to see that $\mathsf{Adv}^{\mathrm{mkror}}_{\mathsf{SE}}(\mathcal{B}, t) = \frac{1}{2} \cdot (\mathsf{Adv}^{\mathrm{mkror}}_{\mathsf{SE}}(\mathcal{B}_1, t) + \mathsf{Adv}^{\mathrm{mkror}}_{\mathsf{SE}}(\mathcal{B}_2, t))$, and so it follows that

$$\begin{aligned}&|\Pr[\,\mathrm{G}_2 \Rightarrow 1\,] - \Pr[\,\mathrm{G}_3 \Rightarrow 1\,]| + \Pr[\,\mathsf{bad} = \mathsf{true} \text{ in } \mathrm{G}_2\,] \\ &\leq \frac{1}{2} \cdot \Big(\mathsf{Adv}^{\mathrm{mkror}}_{\mathsf{SE}}(\mathcal{B}_1, t) + \mathsf{Adv}^{\mathrm{mkror}}_{\mathsf{SE}}(\mathcal{B}_2, t)\Big) + \Pr[\,\mathsf{bad} = \mathsf{true} \text{ in } \mathrm{G}_3\,] \\ &\leq 2 \cdot \mathsf{Adv}^{\mathrm{mkror}}_{\mathsf{SE}}(\mathcal{B}, t) + \Pr[\,\mathsf{bad} = \mathsf{true} \text{ in } \mathrm{G}_3\,]\ .\end{aligned}$$

Now we can define a game G_4 which replaces all encryptions under the public-key encryption scheme PKE with random ciphertexts, where this transition is bounded by a reduction to the ROR security of PKE. Formally we can define an adversary $\mathcal{C}_1$ in game $\mathrm{ROR}^{\mathcal{C}_1}_{\mathsf{PKE}}$ with challenge bit b' who proceeds as follows: on input pk, $\mathcal{C}_1$ first runs $(w_1, \tilde{w}_1, \ldots, \tilde{w}_n) \leftarrow\!\!\$\ \mathcal{T}$, $\bar{s}_n \leftarrow\!\!\$\ \mathsf{PChecker}[\Pi](w, \tilde{w}_1, \ldots, \tilde{w}_n)$. $\mathcal{C}_1$ submits S and the elements in W to its RoR oracle, chooses random symmetric ciphertexts and salts, and assembles the final state including the public key pk it was given as part of its challenge. $\mathcal{C}_1$ then passes s_n to $\mathcal{A}'$, simulating queries to SH in the natural way, and at the end of the game outputs whatever bit $\mathcal{A}'$ does. Notice that if $b' = 0$ then $\mathcal{C}$ receives real encryptions and this perfectly simulates G_3; otherwise

it perfectly simulates G_4. It follows that,

$$\begin{aligned}
&|\Pr[\,\mathrm{G}_3 \Rightarrow 1\,] - \Pr[\,\mathrm{G}_4 \Rightarrow 1\,]| \\
&\qquad = |\Pr[\,\mathcal{C}_1 \Rightarrow 1 \mid b' = 0\,] - \Pr[\,\mathcal{C}_1 \Rightarrow 1 \mid b' = 1\,]| \\
&\qquad \leq \mathsf{Adv}^{\mathrm{ror}}_{\mathsf{PKE}}(\mathcal{C}_1)\ .
\end{aligned}$$

Furthermore, an analogous argument to that made above implies that we can construct an adversary $\mathcal{C}_2$ in the $\mathrm{ROR}^{\mathcal{C}_2}_{\mathsf{PKE}}$ game against PKE who simulates the final state s_n using its RoR oracle, passes s_n to $\mathcal{A}'$ and outputs 1 if and only if $\mathcal{A}'$ sets the flag bad by guessing one of the cached passwords. It follows that,

$$\Pr[\,\mathsf{bad} = \mathsf{true} \text{ in } \mathrm{G}_3\,] \leq \Pr[\,\mathsf{bad} = \mathsf{true} \text{ in } \mathrm{G}_4\,] + \mathsf{Adv}^{\mathrm{ror}}_{\mathsf{PKE}}(\mathcal{C}_2)\ .$$

With this in place, we may again define an adversary $\mathcal{C}$ in the $\mathrm{ROR}^{\mathcal{C}}_{\mathsf{PKE}}$ game against PKEwho randomly chooses to run either $\mathcal{C}_1$ or $\mathcal{C}_2$, and outputs the same bit that they do. It follows that

$$\begin{aligned}
&|\Pr[\,\mathrm{G}_3 \Rightarrow 1\,] - \Pr[\,\mathrm{G}_4 \Rightarrow 1\,]| + \Pr[\,\mathsf{bad} = \mathsf{true} \text{ in } \mathrm{G}_3\,] \\
&\qquad \leq \frac{1}{2} \cdot \left(\mathsf{Adv}^{\mathrm{ror}}_{\mathsf{PKE}}(\mathcal{C}_1) + \mathsf{Adv}^{\mathrm{ror}}_{\mathsf{PKE}}(\mathcal{C}_2)\right) + \Pr[\,\mathsf{bad} = \mathsf{true} \text{ in } \mathrm{G}_4\,] \\
&\qquad\qquad \leq 2 \cdot \mathsf{Adv}^{\mathrm{ror}}_{\mathsf{PKE}}(\mathcal{C}) + \Pr[\,\mathsf{bad} = \mathsf{true} \text{ in } \mathrm{G}_4\,]\ .
\end{aligned}$$

Notice that in game G_4 all values in the state s_n given to $\mathcal{A}'$ are random and independent of $\mathcal{T}$, and so the state s_n may be perfectly simulated by an adversary $\mathcal{G}$ in game $\mathrm{OFFGUESS}^{\mathcal{G},q}_{\Pi,\mathcal{T}}$; we shall now use a reduction to this game to bound the probability that bad is set in game G_4. $\mathcal{G}$ generates a public / secret key pair $(pk, sk) \leftarrow\!\!\$\ \mathcal{K}$, assembles the remainder of s_n by choosing the appropriate random components, and passes s_n to $\mathcal{A}'$. Now each time $\mathcal{A}'$ makes a new query $(\mathsf{sa}||\tilde{w})$ to SH such that $\mathsf{sa} \in \{\mathsf{sa}_0, \ldots, \mathsf{sa}_t\}$, $\mathcal{G}$ returns a fresh random string to $\mathcal{A}'$, and submits a query of the form $(i, \tilde{w})$ to its Test oracle. Since by construction all salts

are distinct, it follows that if $\mathcal{A}$ makes q queries to SH then $\mathcal{G}$ makes at most q queries to his Test oracle also. Therefore,

$$\begin{aligned}\Pr[\,\mathsf{bad} = \mathsf{true} \text{ in game } \mathrm{G}_4\,] &= \Pr\left[\,\mathrm{OFFGUESS}^{\mathcal{G},q}_{\Pi,\mathcal{T}} \Rightarrow 1\,\right] \\ &\leq \mathsf{Adv}^{\mathrm{offguess}}_{\Pi,\mathcal{T}}(\mathcal{G}, q)\ .\end{aligned}$$

In game G_5 we return SH to responding truthfully to oracle queries. Since these values are no longer set during the construction of challenge state s_n, this does not affect the outcome of the game and so $|\Pr[\,\mathrm{G}_4 \Rightarrow 1\,] = \Pr[\,\mathrm{G}_5 \Rightarrow 1\,]|$.

Finally in game G_6 we return to sampling salts without replacement. An identical argument to that made previously implies that,

$$|\Pr[\,\mathrm{G}_5 \Rightarrow 1\,] - \Pr[\,\mathrm{G}_6 \Rightarrow 1\,]| \leq \frac{(t+1)^2}{2^{\ell_{\mathrm{salt}}+1}}\ .$$

Now G_6 is identical to game $\overline{\mathrm{OFFDIST}}^{\mathcal{A}',q}_{\Pi,\mathcal{T}}$ with challenge bit $b = 1$. Putting all this together, we conclude that,

$$\begin{aligned}\mathsf{Adv}^{\mathrm{offdist}}_{\Pi,\mathcal{T}}(\mathcal{A}) \leq\ & \mathsf{Adv}^{\mathrm{offguess}}_{\Pi,\mathcal{T}}(\mathcal{G}, q) + 2 \cdot \mathsf{Adv}^{\mathrm{mkror}}_{\mathsf{SE}}(\mathcal{B}, t) + \frac{(t+1)^2}{2^{\ell_{\mathrm{salt}}}} \\ & + 2 \cdot \mathsf{Adv}^{\mathrm{rob}}_{\mathsf{SE}}(\mathcal{R}) + 2 \cdot \mathsf{Adv}^{\mathrm{ror}}_{\mathsf{PKE}}(\mathcal{C}) + \frac{(t \cdot (n+1)+1)^2}{2^{(\kappa-1)}}\ .\end{aligned}$$

C.4 Proof of Theorem 4.5.2

Before giving the full proof of Theorem 4.5.2, we begin by proving a useful lemma which we will use in subsequent analysis.

Lemma C.4.1 *Let* $\{p_1, \ldots, p_n\}$ *and* $\{x_1, \ldots, x_n\}$ *be sequences of numbers such that each* $p_i, x_i \in [0, 1]$ *and* $p_1 \geq p_2 \geq \cdots \geq p_n$*. Then*

$$\sum_{i=1}^{n} p_i \cdot x_i \leq \sum_{i=1}^{r} p_i \text{ where } r = \left\lceil \sum_{i=1}^{n} x_i \right\rceil\ .$$

Proof: Since $p_1 \geq \cdots \geq p_n$, the rearrangement inequality [161] implies that $\sum_{i=1}^{n} p_i \cdot x_i$ is maximized when the x_i are such that $x_1 \geq x_2 \geq \cdots \geq x_n$, so to upper bound this sum, we reorder them so this is the case. Notice that since $r = \lceil \sum_{i=1}^{n} x_i \rceil \geq \sum_{i=1}^{r} x_i + \sum_{i=r+1}^{n} x_i$, this implies that,

$$r - \sum_{i=1}^{r} x_i \geq \sum_{i=r+1}^{n} x_i \Rightarrow \sum_{i=1}^{r}(1 - x_i) \geq \sum_{i=r+1}^{n} x_i \ .$$

With this in place, it follows that

$$\sum_{i=1}^{r} p_i \cdot (1 - x_i) \geq p_r \cdot \sum_{i=1}^{r}(1 - x_i) \geq p_r \cdot \sum_{i=r+1}^{n} x_i \geq \sum_{i=r+1}^{n} p_i \cdot x_i \ .$$

The first inequality follows since $p_1 \geq \cdots \geq p_r$. The second inequality since $\sum_{i=1}^{r}(1 - x_i) \geq \sum_{i=r+1}^{n} x_i$. The final inequality follows since $p_r \geq \cdots \geq p_n$. Finally rearranging yields,

$$\sum_{i=1}^{r} p_i \geq \sum_{i=1}^{n} p_i \cdot x_i \ ,$$

as required. ∎

We now proceed to the proof of Theorem 4.5.2.

Proof: We wish to upper bound the maximum advantage of an attacker $\mathcal{G}$ who makes at most q queries to the Test oracle in game $\text{OFFGUESS}^{\mathcal{G},q}_{\Pi,\mathcal{T}}$. Let the sequence of guesses made by $\mathcal{G}$ be denoted $G = \{(\tilde{w}_1, j_1), (\tilde{w}_2, j_2), \ldots, (\tilde{w}_q, j_q)\}$, and recall that if $j_i = 0$ then this corresponds to a guess at the value of the real password; otherwise the guess represents a guess at the typo stored at position $0 < j_i \leq t$ in the cache. Without loss of generality, we may assume that $\mathcal{G}$ never repeats a query, since this would decrease his success probability. We split the guesses into two sets Z_0 and Z_1 where,

$$Z_0 = \{(\tilde{w}_i, j_i) \,|\, j_i = 0\} \text{ and } Z_1 = \{(\tilde{w}_i, j_i) \,|\, 0 < j_i \leq t\} \ .$$

We let $q_0 = |Z_0|$ and $q_1 = |Z_1|$, so $q_1 \leq q - q_0$. We let $T[j]$ denote the distribution of

the typo at the j^{th} position in the cache. Notice that the adversary $\mathcal{G}$ will succeed if either the real password $T[0]$ lies in the set Z_0, or $T[j_i] = \tilde{w}_i$ for some $(\tilde{w}_i, j_i) \in Z_1$. It follows that

$$\begin{aligned}\mathsf{Adv}^{\text{offguess}}_{\Pi,\mathcal{T}}(\mathcal{G}, q) &= \Pr[T[0] \in Z_0 \ \vee \ \exists(\tilde{w}_i, j_i) \in Z_1 \ : \ T[j_i] = \tilde{w}_i] \\ &= \Pr[T[0] \in Z_0] + \Pr[T[0] \notin Z_0 \wedge \exists(\tilde{w}_i, j_i) \in Z_1 \ : \ T[j_i] = \tilde{w}_i] \ .\end{aligned}$$

We may rewrite the above expression

$$\begin{aligned}&\mathsf{Adv}^{\text{offguess}}_{\Pi,\mathcal{T}}(\mathcal{G}, q) - \Pr[T[0] \in Z_0] \\ &= \sum_{w \in \mathcal{P} \setminus Z_0} \Pr[\exists(\tilde{w}_i, j_i) \in Z_1 \ : \ T[j_i] = \tilde{w}_i \,|\, T[0] = w] \cdot \Pr[T[0] = w] \\ &= \sum_{w \in \mathcal{P} \setminus Z_0} \Pr\Big[\bigvee_{(\tilde{w}_i, j_i) \in Z_1} T[j_i] = \tilde{w}_i \,|\, T[0] = w\Big] \cdot \Pr[T[0] = w] \ .\end{aligned}$$

For each $w \in \mathcal{P}$, $\Pr\big[\bigvee_{(\tilde{w}_i, j_i)} T[j_i] = \tilde{w}_i \mid T[0] = w\big] \in [0, 1]$, and so an application of Lemma C.4.1 implies that

$$\begin{aligned}\mathsf{Adv}^{\text{offguess}}_{\Pi,\mathcal{T}}(\mathcal{G}, q) &\leq \sum_{w \in Z_0} \Pr[T[0] = w] + \sum_{w \in Z_1^*} \Pr[T[0] = w] \\ &\leq \sum_{i=1}^{q_0 + \lceil q_1' \rceil} \Pr[T[0] = w_i] \ ,\end{aligned}$$

where Z_1^* is the set of the q_1' heaviest passwords in $\mathcal{P} \setminus Z_0$ and,

$$q_1' = \left\lceil \sum_{w \in \mathcal{P} \setminus Z_0} \Pr\Big[\bigvee_{(\tilde{w}_i, j_i) \in Z_1} T[j_i] = \tilde{w}_i \mid T[0] = w\Big] \right\rceil \ .$$

We now upper bound q_1'. Since by assumption the error setting is t-sparse, it holds

that $b_{\tilde{\tau}}(\tilde{w}) \leq t$ for all $\tilde{w} \in \mathcal{M}$. It follows that

$$\begin{aligned}
q_1' &\leq \sum_{w \in \mathcal{P} \setminus Z_0} \Pr\Big[\bigvee_{(\tilde{w}_i, j_i) \in Z_1} T[j_i] = \tilde{w}_i \mid T[0] = w\Big] \\
&\leq \sum_{w \in \mathcal{P}} \Pr\Big[\bigvee_{(\tilde{w}_i, j_i) \in Z_1} T[j_i] = \tilde{w}_i \mid T[0] = w\Big] \\
&\leq \sum_{w \in \mathcal{P}} \sum_{(\tilde{w}_i, j_i) \in Z_1} \Pr[T[j_i] = \tilde{w}_i \mid T[0] = w] \\
&= \sum_{w \in \mathcal{P}} \sum_{(\tilde{w}_i, j_i) \in Z_1} \frac{1}{t} \cdot \tilde{\tau}_w(\tilde{w}_i) \\
&= \sum_{(\tilde{w}_i, j_i) \in Z_1} \frac{1}{t} \cdot b_{\tilde{\tau}}(\tilde{w}_i) \quad \leq q - q_0 \ .
\end{aligned}$$

The first inequality follows since $\mathcal{P} \setminus Z_0 \subseteq \mathcal{P}$ for any Z_0. The second inequality follows by taking a union bound over the points in Z_1. The next equality follows because the typo cache elements are distinct and randomly permuted, so $\Pr[T[j_i] = \tilde{w}_i \mid T[0] = w] = \frac{1}{t} \cdot \tilde{\tau}_w(\tilde{w})$. The next equality follows since by definition $b_{\tilde{\tau}}(\tilde{w}_i) = \sum_{w \in \mathcal{P}} \tilde{\tau}_w(\tilde{w}_i)$. The final inequality follows since $b_{\tilde{\tau}}(\tilde{w}_i) \leq t$ for all $\tilde{w}_i$, and there are at most $q - q_0$ guesses in Z_1. Putting this all together implies that,

$$q_0 + q_1' \leq q_0 + (q - q_0) = q \ ,$$

and we conclude that

$$\mathsf{Adv}^{\text{offguess}}_{\Pi, \mathcal{T}}(\mathcal{G}, q) \leq \sum_{i=1}^{q} p(w_i) \ . \qquad \blacksquare$$

C.5 Online Security

Following from the discussion in Section 4.5.3, we now detail the security analysis of TypTop in the online setting.

We define online security via the game ONGUESS depicted in Figure C.2,

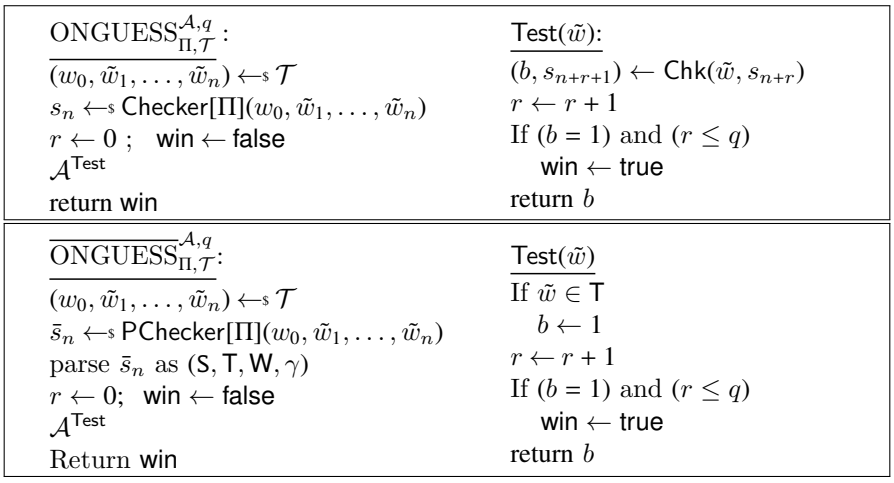


Figure C.2: Security games for online attacks.

adapting the corresponding notion formulated by Chatterjee et al. in [30] to the adaptive checking setting, with the advantage defined

$$\mathsf{Adv}^{\text{onguess}}_{\Pi,\mathcal{T}}(\mathcal{A}, q) = \Pr\left[\,\text{ONGUESS}^{\mathcal{A},q}_{\Pi,\mathcal{T}} \Rightarrow \mathsf{true}\,\right].$$

We sample a password and login transcript via the transcript generator $\mathcal{T}$ and evolve the state of the adaptive checker accordingly. The attacker is given access to an oracle Test to which he may submit guesses; he succeeds if he makes a guess which is accepted by the checking algorithm Chk. The game is parameterized by q representing the number of Test queries $\mathcal{A}$ is allowed; this reflects the standard online attack countermeasure of locking an account after a certain number of incorrect guesses.

The analysis. Following the similar discussion in Section 4.5, we first define a game $\overline{\text{ONGUESS}}$ analogous to $\overline{\text{OFFGUESS}}$, in which the final cache state is generated via the plaintext checker PChecker, and the advantage is defined as

$$\mathsf{Adv}^{\overline{\text{onguess}}}_{\Pi,\mathcal{T}}(\mathcal{A}, q) = \Pr\left[\,\overline{\text{ONGUESS}}^{\mathcal{A},q}_{\Pi,\mathcal{T}} \Rightarrow \mathsf{true}\,\right].$$

In Lemma C.5.1 we bound the difference between the two games for TypTop in terms of the robustness of the underlying SE scheme SE.

Lemma C.5.1 *Let (p, τ) be an error setting with associated transcript generator $\mathcal{T}$, and let $\Pi = (\mathit{Reg}, \mathit{Chk})$ be TypTop's password checker with associated plaintext checker $\mathsf{PChecker}[\Pi]$. Let Π be implemented using the canonical PBE scheme $\mathsf{PBE}[\mathsf{SH}, \mathsf{SE}] = (\overline{\mathsf{E}}, \overline{\mathsf{D}})$ where SE is a symmetric encryption scheme and SH is a random oracle. Then for any adversary $\mathcal{A}$ running in time T, there exist adversaries $\mathcal{A}'$, $\mathcal{R}$ such that*

$$\mathsf{Adv}^{\mathrm{onguess}}_{\Pi,\mathcal{T}}(\mathcal{A}, q) \leq \mathsf{Adv}^{\overline{\mathrm{onguess}}}_{\Pi,\mathcal{T}}(\mathcal{A}', q) + \mathsf{Adv}^{\mathrm{rob}}_{\mathsf{SE}}(\mathcal{R}) + \frac{(t \cdot (n + 1 + q) + 1 + q)^2}{2^{\kappa}} ,$$

and, moreover, $\mathcal{A}'$ runs in time $T' \approx T$. Here t denotes the size of the cache, n denotes the length of the transcript output by $\mathcal{T}$ and SE has key space $\{0,1\}^{\kappa}$.

Proof: We argue by a series of game hops. Let game G_0 be equivalent to game $\mathrm{ONGUESS}_{\Pi,\mathcal{T}}$, so

$$\mathsf{Adv}^{\mathrm{onguess}}_{\Pi,\mathcal{T}}(\mathcal{A}, q) = \Pr[\,\mathrm{G}_0 \Rightarrow 1\,] .$$

Let game G_1 be identical to G_0 except that the keys used to compute the cached ciphertexts in state s_n, and those used for trial decryptions in response to Test queries made by $\mathcal{A}$ in the guessing stage of the game while $\mathsf{win} = \mathsf{false}$, are sampled without replacement. These games run identically unless two of the keys sampled during these phases collide. There are at most $(t \cdot (n+1)+1)$ such keys sampled while computing the cached ciphertexts for a transcript of length n, and at most $q \cdot (t+1)$ keys sampled during the guessing phase (reflecting the $(t + 1)$ trial decryptions performed by Chk for each of the q Test queries made by $\mathcal{A}$). Notice that since cache updates only occur in the guessing phase if $\mathcal{A}$ guesses a string which is accepted by Chk, the cache will never update while $\mathsf{win} = \mathsf{false}$. It follows that

$$|\Pr[\,\mathrm{G}_0 \Rightarrow 1\,] - \Pr[\,\mathrm{G}_1 \Rightarrow 1\,]| \leq \frac{(t \cdot (n + 1 + q) + 1 + q)^2}{2^{\kappa+1}} .$$

Next we define game G_2 which is identical to G_1 except that we replace $\mathsf{Checker}[\Pi]$ with $\mathsf{PChecker}$ and redefine Test to perform comparisons on the plaintext typo cache output by $\mathsf{PChecker}$. Since the adversary in these games never sees the internal state of the checker, not encrypting the values which lie in this state does not change the adversary's view of the game; rather the two run identically unless during the process of updating the state and the adversary's subsequent queries to Test we find two distinct keys $\mathsf{k}_1 \neq \mathsf{k}_2$ such that $\mathsf{D}_{\mathsf{k}_2}(\mathsf{E}_{\mathsf{k}_1}(sk)) \neq \perp$ where sk denotes the secret key of the PKE scheme which is encrypted under each of the cached typos. Thus the fundamental lemma of game playing implies that the gap between game G_1 and G_2 is upper bounded by the probability that this event occurs. Consider an adversary $\mathcal{R}$ in game $\mathrm{ROB}^{\mathcal{R}}_{\mathsf{SE}}$ who simply executes the game G_1, simulating SH by sampling random strings without replacement, and checking if there ever exists a typo cache ciphertext $\mathsf{E}_{\mathsf{k}_1}(sk)$ that decrypts under some subsequently sampled $\mathsf{k}_2 \neq \mathsf{k}_1$ (recall that since G_1 samples without replacement, all keys are distinct). The robustness of the encryption scheme implies that the probability that this event occurs, and thus the gap between games G_1 and G_2 is upper-bounded by $\mathsf{Adv}^{\mathrm{rob}}_{\mathsf{SE}}(\mathcal{R})$

$$|\Pr[\,G_1 \Rightarrow 1\,] - \Pr[\,G_2 \Rightarrow 1\,]| \leq \mathsf{Adv}^{\mathrm{rob}}_{\mathsf{SE}}(\mathcal{R})\ .$$

Next we define game G_3 which is identical to G_2 except we return SH to sampling keys with replacement. An analogous argument to that above bounding the probability that two keys collide ensures that

$$|\Pr[\,G_2 \Rightarrow 1\,] - \Pr[\,G_3 \Rightarrow 1\,]| \leq \frac{(t \cdot (n+1+q)+1+q)^2}{2^{\kappa+1}}\ .$$

Notice that G_3 is identical to $\overline{\mathrm{ONGUESS}}^{\mathcal{A}}_{\Pi,\mathcal{T}}$, and so can be perfectly simulated by an adversary $\mathcal{A}'$ in this game. $\mathcal{A}'$ simulates $\mathcal{A}$'s Test oracle by submitting $\mathcal{A}$'s queries to his own oracle, and returning the responses. Since $\mathcal{A}'$ makes precisely the same set of queries as $\mathcal{A}$, it follows that if $\mathcal{A}$ makes at most q queries, then $\mathcal{A}'$

does also. Since both games are identically distributed, it follows that

$$\mathsf{Adv}^{\overline{\text{onguess}}}_{\Pi,\mathcal{T}}(\mathcal{A}', q) = \Pr[\,\mathrm{G}_3 \Rightarrow 1\,]\,,$$

concluding the proof. ∎

Online guessing advantage. It remains to bound $\mathsf{Adv}^{\overline{\text{onguess}}}_{\Pi,\mathcal{T}}(\mathcal{A}, q)$. The key difference between the online guessing game and its offline counterpart is that in the former each guess is tested for equality against each of the $t + 1$ positions in the cache, whereas in the latter a guess is only checked against the specific slot to which it was guessed.

We reduce the online guessing game — in which the attacker's goal is to find q guesses that maximizes its success probability — to a weighted maximum coverage problem, and use an approximate greedy algorithm to compute the attacker's advantage. We can then bound the advantage of the optimal attacker using the classic result of [100]. However, due to the complex dependencies of the cached elements, we could not show the reduction in the other direction: that is to say, reduce an NP-complete problem to that of finding the optimal set of guesses in the online guessing game. We strongly believe that this problem is NP-hard but leave the detailed reduction as an open problem.

Approximation via greedy algorithm. Recall that the maximum coverage problem is defined as follows. Given n subsets S_i from a universe U, the goal is to find k subsets that cover the maximum number of elements. In the weighted version of this problem, every element in U is weighted, and the goal is to maximize the sum total weight of the covered elements.

We reduce the online guessing game $\overline{\text{ONGUESS}}$ for a particular error setting

(p, τ) and plaintext checker $\mathsf{PChecker}[\Pi]$ to a weighted maximum coverage problem as follows. We define the universe U to be the set of all possible cache-tuples, where a cache-tuple consists of a password $w \in \mathcal{P}$ followed by at most t distinct and alphabetically sorted typos $\tilde{w}_i \in \mathcal{S}$. The weight of a given cache-tuple is defined to be the probability that this tuple lies in the cache of the state $\bar{s}_n$ that the attacker guesses against in game $\overline{\text{ONGUESS}}$. For each password or typo $\tilde{w}$, we define $S_{\tilde{w}} \subseteq U$ to be the set of all cache-tuples that contain $\tilde{w}$. Given all such subsets, the attacker's goal is to find q subsets so that the sum total of the covered cached-tuples is maximized.

With this reduction in place, we can apply the greedy approximation algorithm for finding the weighted maximum coverage.

Empirical analysis. We wish to compute the advantage of an adversary in the online guessing game for real world error settings. While it is easy to describe the reduction to a weighted maximum coverage problem, generating the universe of cache-tuples and the corresponding subsets for large numbers of passwords and typos is computationally very expensive. For example, there could be more than a billion cache-tuples for a password with 100 typos and cache size $t = 5$, and finding all such cache-tuples for a large number of passwords seems infeasible.

We therefore perform the simulation on a subset of k passwords from RockYou in the following way. For each real password w, we sample m typos from τ_w with replacement, run the plaintext checker $\mathsf{PChecker}[\Pi]$ on the sampled list, and record the final cache-tuple. We repeat this process n times for each password, and record all the unique cache-tuples with their weight set to $p(w) \cdot f/n$, where f is the number of times the cache-tuple was observed. We set the universe U to be the set of all cache-tuples we collected in the above experiment, and for each string

$\tilde{w}$, subset $S_{\tilde{w}}$ is defined as the set of all cache-tuples from U which contain $\tilde{w}$. The attacker's goal is to find a set of q strings $\tilde{w}$ such that the cumulative weight of the elements covered by their subsets is maximized.

The greedy algorithm to find the weighted maximum cover works as follows: find the subset $S_{\tilde{w}^*}$ that has the highest cumulative weight, add the corresponding string $\tilde{w}^*$ to the list of guesses, remove all occurrences of cache-tuples in $S_{\tilde{w}^*}$ from other subsets, and repeat until q guesses are found or all subsets are empty.

We wish to compute the security loss incurred by using TypTop compared to an exact checker. Recall that λ_q denotes the success probability of an optimal attack against an exact checker with a budget of q guesses, and that $\lambda_q = \sum_{i=1}^{q} p(w_i)$. We define the security loss of a checker Π over the exact checker to be

$$\Delta_q = \mathsf{Adv}^{\overline{\text{onguess}}}_{\Pi,\mathcal{T}}(\mathcal{A}, q) - \lambda_q \ .$$

Using k most frequent passwords from RockYou, we ran the above simulation with $m = 200$, $n = 500$. We chose $k = 10^5$, and for each caching policy we compute the greedy attacker's advantage for $q = 100$. For all caching policies the security loss Δ_q is minimal, with a maximum security loss of $\Delta_{100} = 0.001$ for the MFU caching policy, and less than 0.0006 for all other caching policies. We also tried sampling passwords randomly from the support of the password distribution, and taking the k most frequent passwords in RockYou after ignoring the first million passwords. The security loss is even less in such samples as we observed in the offline scenario in Section 4.5.2. To see the effect of n on the final security loss, we also ran the experiment with $n = 1000$ for the PLFU caching strategy. We found negligible change in the security loss.

By the result of Hochbaum [100], we know that the output of the greedy algorithm is no less than $1 - 1/e$ times that of the optimal algorithm. If we include

this adjustment into the output of our greedy approximation algorithm, we get $\Delta_q \leq \frac{e}{e-1} \cdot \mathsf{Adv}^{\overline{\text{onguess}}}_{\Pi,\mathcal{T}}(\mathcal{A}, q) - \lambda_q$. Therefore, the security loss due to TypTop is at most $\Delta_{100} \leq 1.582 \times 0.0456 - 0.045 = 0.027$.

This bound is pessimistic. It assumes the attacker has precise knowledge of the typo distribution. Moreover, the final bound is looser if the greedy approximation results are closer to the optimal—which we believe to be the case.

We might be able to use a blacklisting strategy similar to the one proposed in [30] to further reduce the security loss. A naive blacklisting approach would be to block a set of 'risky' typos (that is to say those which allow an attacker to achieve too great an advantage) from entering the typo-cache. However to decide which typos to blacklist, we need an accurate measure of the cache inclusion function, which will itself change each time a new typo is blacklisted, significantly complicating the analysis of this approach. We leave a detailed treatment of blacklisting strategies for future work.

APPENDIX D

APPENDIX - CHECKING LEAKED PASSWORDS

D.1 Bandwidth and Security of FSB

$$
\begin{aligned}
m &= \sum_{w \in \tilde{\mathcal{S}}} |\beta(w)| \\
&= \sum_{w \in \mathcal{P}_{\bar{q}} \cap \tilde{\mathcal{S}}} |\mathcal{B}| + \sum_{w \in \tilde{\mathcal{S}} \setminus \mathcal{P}_{\bar{q}}} \left\lceil \frac{|\mathcal{B}| \cdot \hat{p}_s(w)}{\hat{p}_s(w_{\bar{q}})} \right\rceil \\
&\leq |\mathcal{P}_{\bar{q}} \cap \tilde{\mathcal{S}}| \cdot |\mathcal{B}| + \sum_{w \in \tilde{\mathcal{S}} \setminus \mathcal{P}_{\bar{q}}} \left(\frac{|\mathcal{B}| \cdot \hat{p}_s(w)}{\hat{p}_s(w_{\bar{q}})} + 1 \right) \\
&\leq |\mathcal{B}| \cdot \bar{q} + |\mathcal{B}| \cdot \frac{1}{\hat{p}_s(w_{\bar{q}})} + N
\end{aligned}
$$

The first equality is obtained by replacing the definition of $\beta(w)$; the second inequality holds because $\lceil x \rceil \leq x + 1$; the third inequality holds because $S \subseteq W$.

$n = |B|$; and $m > n \log n$, if $\bar{q} > \log n$. Therefore, the maximum bucket size for FSB would be no more than $2 \cdot \left(\bar{q} + \frac{1}{\hat{p}_s(w_{\bar{q}})} + \frac{N}{|\mathcal{B}|} \right)$.

D.2 Correlation between username and passwords

In Section 5.3 the username and password choices of previously uncompromised users can be modeled independently.

To check whether this assumption would be valid or not, we randomly sampled 10^5 username-password pairs from the dataset used in Section 5.6 and calculated the Levenshtein edit distance between each username and password in a pair. We have recorded the result of this experiment in Figure D.1.

Distance	%
0	1.2
≤ 1	1.7
≤ 2	2.3
≤ 3	3.1
≤ 4	4.6

Figure D.1: Statistics on samples with low edit distance between username and password, as a percentage of a random sample of 10^5 username-password pairs.

We found that the mean edit distance between a username and password was 9.4, while the mean password length was 8.4 characters and the mean username length was 10.0 characters. This supports that while there are some pairs where the password is almost identical to the username, a large majority are not related to the username at all.

D.3 Proof of Theorem 5.4.2

Because the IDB bucketization scheme does not depend on the password, $\Pr[\, B = b \mid W = w \wedge U = u \,] = \Pr[\, B = b \mid \wedge U = u \,]$

$$
\begin{aligned}
&\mathsf{Adv}^{\text{b-gs}}_{\text{IDB}}(q) \\
&= \sum_u \sum_b \max_{w_1,\ldots,w_q} \sum_{i=1}^{q} \Pr[\, W = w_i \wedge U = u \,] \cdot \Pr[\, B = b \mid U = u \,] \\
&= \sum_u \left(\sum_b \Pr[\, B = b \mid U = u \,] \right) \max_{w_1,\ldots,w_q} \sum_{i=1}^{q} \Pr[\, W = w_i \wedge U = u \,] \\
&= \mathsf{Adv}^{\text{gs}}(q)
\end{aligned}
$$

The first step follows from independence of password and bucket choice, and the third step is true because there is only one bucket for each username.

D.4 Proof of Theorem 5.5.1

$$
\begin{aligned}
&\mathsf{Adv}^{\text{b-gs}}_{\beta_{\text{FSB}}}(q) \\
&= \sum_u \sum_b \max_{\substack{w_1,\ldots,w_q \\ \in \alpha(b)}} \sum_{i=1}^{q} \frac{\Pr[\, W = w_i \wedge U = u \,]}{|\beta_{\text{FSB}}(w_i)|} \\
&= \sum_b \max_{\substack{w_1,\ldots,w_q \\ \in \alpha(b)}} \sum_{i=1}^{q} \frac{\hat{p}_s(w_i)}{|\beta_{\text{FSB}}(w_i)|}
\end{aligned}
$$

The second step follows from the independence of usernames and passwords in the uncompromised setting.

We will use $\mathcal{P}_{\bar{q}}$ to refer to the top $\bar{q}$ passwords according to password distribution $\hat{p}_s = p_w$, and $w_{\bar{q}}$ to refer to the $\bar{q}$th most popular password according to $\hat{p}_s$.

For any $w \in \mathcal{P}_{\bar{q}}$, $\frac{\hat{p}_s(w)}{|\beta_{\text{FSB}}(w)|} = \frac{\hat{p}_s(w)}{|\mathcal{B}|}$.

For any $w \in \mathcal{P} \setminus \mathcal{P}_{\bar{q}}$,

$$
\frac{|\mathcal{B}| \cdot \hat{p}_s(w)}{\hat{p}_s(w_{\bar{q}})} \leq |\beta_{\text{FSB}}(w)| < \frac{|\mathcal{B}| \cdot \hat{p}_s(w)}{\hat{p}_s(w_{\bar{q}})} + 1.
$$

We can use the lower bound on $|\beta_{\text{FSB}}(w)|$ to find that

$$
\frac{\hat{p}_s(w)}{|\beta_{\text{FSB}}(w)|} \leq \frac{\hat{p}_s(w_{\bar{q}})}{|\mathcal{B}|}.
$$

Using the upper bound on $|\beta_{\text{FSB}}(w)|$,

$$
\frac{\hat{p}_s(w)}{|\beta_{\text{FSB}}(w)|} > \frac{\hat{p}_s(w)}{\frac{|\mathcal{B}| \cdot \hat{p}_s(w)}{\hat{p}_s(w_{\bar{q}})} + 1} = \frac{\hat{p}_s(w) \cdot \hat{p}_s(w_{\bar{q}})}{|\mathcal{B}| \cdot \hat{p}_s(w) + \hat{p}_s(w_{\bar{q}})} = \frac{\hat{p}_s(w_{\bar{q}})}{|\mathcal{B}| + \frac{\hat{p}_s(w_{\bar{q}})}{\hat{p}_s(w)}}
$$

Since the values of $\frac{\hat{p}_s(w)}{|\beta_{\text{FSB}}(w)|}$ are always larger for $w \in \mathcal{P}_{\bar{q}}$, the values of $w_1, \ldots, w_q$ chosen for each bucket will be the top $\bar{q}$ passwords overall, along with the top $q - \bar{q}$ of the remaining passwords in the bucket, ordered by $\frac{\hat{p}_s(\cdot)}{|\beta_{\text{FSB}}(\cdot)|}$.

To find an upper bound on $\mathsf{Adv}^{\text{b-gs}}_{\beta_{\text{FSB}}}(q)$,

$$\begin{aligned}
&\sum_b \max_{\substack{w_1,\ldots,w_q \\ \in\alpha(b)}} \sum_{i=1}^{q} \frac{\hat{p}_s(w_i)}{|\beta_{\text{FSB}}(w_i)|} \\
&\leq \sum_b \left(\sum_{w\in\mathcal{P}_{\bar{q}}} \frac{\hat{p}_s(w)}{|\mathcal{B}|} + (q-\bar{q})\frac{\hat{p}_s(w_{\bar{q}})}{|\mathcal{B}|} \right) \\
&= \lambda_{\bar{q}} + (q-\bar{q}) \cdot p_{\bar{q}}
\end{aligned}$$

For $q \leq \bar{q}$, we have $\mathsf{Adv}^{\text{b-gs}}_{\beta_{\text{FSB}}}(q) \leq \lambda_{\bar{q}}$.

To find a lower bound on $\mathsf{Adv}^{\text{b-gs}}_{\beta_{\text{FSB}}}(q)$, let $w^*_{\bar{q}+1}, \ldots, w^*_q$ be the $q-\bar{q}$ passwords in $\alpha(b) \setminus \mathcal{P}_{\bar{q}}$ with the highest probability of occurring, according to $\hat{p}_s(\cdot)$.

$$\begin{aligned}
&\sum_b \max_{\substack{w_1,\ldots,w_q \\ \in\alpha(b)}} \sum_{i=1}^{q} \frac{\hat{p}_s(w_i)}{|\beta_{\text{FSB}}(w_i)|} \\
&> \sum_b \left(\sum_{w\in\mathcal{P}_{\bar{q}}} \frac{\hat{p}_s(w)}{|\mathcal{B}|} + \sum_{i=\bar{q}+1}^{q} \frac{\hat{p}_s(w_{\bar{q}})}{|\mathcal{B}| + \frac{\hat{p}_s(w_{\bar{q}})}{\hat{p}_s(w^*_i)}} \right) \\
&\geq \lambda_{\bar{q}} + \sum_{i=\bar{q}+1}^{q} \left\lceil \frac{|\mathcal{B}| \cdot \hat{p}_s(w^*_i)}{\hat{p}_s(w_{\bar{q}})} \right\rceil \cdot \frac{\hat{p}_s(w_{\bar{q}})}{|\mathcal{B}| + \frac{\hat{p}_s(w_{\bar{q}})}{\hat{p}_s(w^*_i)}} \\
&\geq \lambda_{\bar{q}} + \sum_{i=\bar{q}+1}^{q} \frac{|\mathcal{B}| \cdot \hat{p}_s(w^*_i)}{|\mathcal{B}| + \frac{\hat{p}_s(w_{\bar{q}})}{\hat{p}_s(w^*_i)}} \geq \lambda_{\bar{q}} + \sum_{i=\bar{q}+1}^{q} \frac{\hat{p}_s(w^*_i)}{1 + \frac{\hat{p}_s(w_{\bar{q}})}{\hat{p}_s(w^*_i)\cdot|\mathcal{B}|}} \\
&\geq \lambda_{\bar{q}} + \sum_{i=\bar{q}+1}^{q} \hat{p}_s(w^*_i)/2 \geq \lambda_{\bar{q}} + (\lambda_q - \lambda_{\bar{q}})/2 = \frac{\lambda_q + \lambda_{\bar{q}}}{2}
\end{aligned}$$

Therefore, $\Delta_q \geq \frac{\lambda_q - \lambda_{\bar{q}}}{2}$.

Note, for every password to be assigned to a bucket, $|\mathcal{B}| \geq \hat{p}_s(w_{\bar{q}})/\hat{p}_s(w)$, or for all $w \in \mathcal{P}$, $\frac{\hat{p}_s(w_{\bar{q}})}{\hat{p}_s(w)\cdot|\mathcal{B}|} \leq 1$.

BIBLIOGRAPHY

[1] Have I been pwned: API v2. https://haveibeenpwned.com/API/v2, 2018.

[2] Password check. https://security.googleblog.com/2019/02/protect-your-accounts-from-data.html, 2018.

[3] Touch ID. https://en.wikipedia.org/wiki/Touch_ID, 2018.

[4] Face ID. https://en.wikipedia.org/wiki/Face_ID, 2018.

[5] YubiKey. https://en.wikipedia.org/wiki/YubiKey, 2016. Accessed: 2019-06-09.

[6] OAuth. https://en.wikipedia.org/wiki/OAuth, 2010. Accessed: 2019-06-09.

[7] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In *2012 IEEE Symposium on Security and Privacy*, May 2012.

[8] **passwd**. https://en.wikipedia.org/wiki/Passwd.

[9] Danny Yadron. Man behind the first computer password: Its become a nightmare. https://blogs.wsj.com/digits/2014/05/21/the-man-behind-the-first-computer-password-its-become-a-nightmare/, 2014.

[10] R. Morris and K. Thompson. Password security: a case history. *Commun. ACM*, 22(11):594–597, November 1979.

[11] M-209. https://en.wikipedia.org/wiki/M-209.

[12] Niels Provos and David Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.

[13] Colin Percival. Stronger key derivation via sequential memory-hard functions. 2009.

[14] 4iQ. Identities in the Wild: The Tsunami of Breached Identities Continues. https://4iq.com/wp-content/uploads/2018/05/2018_IdentityBreachReport_4iQ.pdf/, 2018.

[15] Andy Greenberg. Hackers are passing around a megaleak of 2.2 billion records. https://www.wired.com/story/collection-leak-usernames-passwords-billions/, Jan, 2019.

[16] Munir Kotadia. Gates predicts death of the password. https://www.cnet.com/news/gates-predicts-death-of-the-password/, February 2004.

[17] A. Czeskis, M. Dietz, T. Kohno, D. Wallach, and D. Balfanz. Strengthening user authentication through opportunistic cryptographic identity assertions. In *ACM Conference on Computer and Communications Security (ACM CCS)*, pages 404–414, 2012.

[18] IH Jermyn, Alain Mayer, Fabian Monrose, Michael K Reiter, and Aviel D Rubin. The design and analysis of graphical passwords. USENIX Association, 1999.

[19] Saranga Komanduri, Richard Shay, Patrick Gage Kelley, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Serge Egelman. Of passwords and people: Measuring the effect of password-composition policies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2595–2604, New York, NY, USA, 2011. ACM.

[20] Richard Shay, Saranga Komanduri, Patrick Gage Kelley, Pedro Giovanni Leon, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Encountering stronger password requirements: User attitudes and behaviors. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, SOUPS '10, pages 2:1–2:20, New York, NY, USA, 2010. ACM.

[21] Paul A Grassi, JL Fenton, EM Newton, RA Perlner, AR Regenscheid, WE Burr, JP Richer, NB Lefkovitz, JM Danker, YY Choong, et al. Nist special publication 800-63b. digital identity guidelines: Authentication and lifecycle management. *Bericht, NIST*, 2017.

[22] Lastpass. The password exposé. https://www.lastpass.com/state-of-the-password, 2018. Accessed on: May, 2019.

[23] Tom Le Bras. Online overload – It's worse than you thought. https://blog.dashlane.com/infographic-online-overload-its-worse-than-you-thought/, 2015. Accessed on: May, 2019.

[24] P.G. Kelley, S. Komanduri, M.L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L.F. Cranor, and J. Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *IEEE Symposium on Security and Privacy (SP)*, pages 523–537, 2012.

[25] A. Juels and T. Ristenpart. Honey Encryption: Beyond the brute-force barrier. In *Advances in Cryptology – EUROCRYPT*, pages 523–540. Springer, 2014.

[26] Kenneth Olmstead and Aaron Smith. What the public knows about cybersecurity. *PewResearchCenter, March*, 22, 2017.

[27] Sarah Pearman, Jeremy Thomas, Pardis Emami Naeini, Hana Habib, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Serge Egelman, and Alain Forget. Let's go in for a closer look: Observing passwords in their natural habitat. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 295–310. ACM, 2017.

[28] Verizon Enterprise. 2017 data breach investigations report, 2017.

[29] Rahul Chatterjee, Joseph Bonneau, Ari Juels, and Thomas Ristenpart. Cracking-resistant password vaults using natural language encoders. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 481–498. IEEE, 2015.

[30] Rahul Chatterjee, Anish Athalye, Devdatta Akhawe, Ari Juels, and Thomas Ristenpart. password typos and how to correct them securely. *IEEE Symposium on Security and Privacy*, may 2016.

[31] Rahul Chatterjee, Joanne Woodage, Yuval Pnueli, Anusha Chowdhury, and Thomas Ristenpart. The typtop system: Personalized typo-tolerant password checking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 329–346. ACM, 2017.

[32] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. *arXiv preprint arXiv:1905.13737*, 2019.

[33] Lance Whitney. LastPass CEO reveals details on security breach. *CNet*, May 2011.

[34] Zhiwei Li, Warren He, Devdatta Akhawe, and Dawn Song. The emperor's new password manager: Security analysis of web-based password managers. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[35] Burt Kaliski. PKCS #5: Password-based cryptography specification version 2.0, 2000. RFC 2289.

[36] Joseph Bonneau. *Guessing human-chosen secrets*. PhD thesis, University of Cambridge, May 2012.

[37] Hristo Bojinov, Elie Bursztein, Xavier Boyen, and Dan Boneh. Kamouflage: Loss-resistant password management. In *European symposium on research in computer security*, pages 286–302. Springer, 2010.

[38] D.N. Hoover and B.N. Kausik. Software smart cards via cryptographic camouflage. In *IEEE Symposium on Security and Privacy*, pages 208–215. IEEE, 1999.

[39] M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek. Password cracking using probabilistic context-free grammars. In *IEEE Symposium on Security and Privacy (SP)*, pages 162–175, 2009.

[40] Rafael Veras, Christopher Collins, and Julie Thorpe. On the semantic patterns of passwords and their security impact. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[41] Jerry Ma, Weining Yang, Min Luo, and Ninghui Li. A study of probabilistic password models. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP)*, pages 689–704. IEEE Computer Society, 2014.

[42] Martin E Hellman. A cryptanalytic time-memory trade-off. *Information Theory, IEEE Transactions on*, 26(4):401–406, 1980.

[43] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Advances in Cryptology-CRYPTO 2003*, pages 617–630. Springer, 2003.

[44] M. Bellare, T. Ristenpart, and S. Tessaro. Multi-instance security and its application to password-based cryptography. In *Advances in Cryptology – CRYPTO 2012*, pages 312–329. Springer Berlin Heidelberg, 2012.

[45] Joseph Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *IEEE Symposium on Security and Privacy (SP)*, pages 538–552. IEEE, 2012.

[46] Markus Jakobsson and Mayank Dhiman. The benefits of understanding passwords. In *Mobile Authentication*, pages 5–24. Springer, 2013.

[47] Claude Castelluccia, Markus Dürmuth, and Daniele Perito. Adaptive password-strength meters from markov models. In *NDSS*, 2012.

[48] Martin MA Devillers. Analyzing password strength. *Radboud University Nijmegen, Tech. Rep*, 2010.

[49] Sylvie Billot and Bernard Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th Annual Meeting on Association for Computational Linguistics*, ACL '89, pages 143–151, Stroudsburg, PA, USA, 1989. Association for Computational Linguistics.

[50] Grune Dick and H Ceriel. Parsing techniques, a practical guide. Technical report, Technical Report, 1990.

[51] Leo Breiman. Random Forests. *Machine learning*, 45(1):5–32, 2001.

[52] Luis Von Ahn, Manuel Blum, Nicholas J Hopper, and John Langford. CAPTCHA: Using hard AI problems for security. In *Advances in Cryptology—EUROCRYPT 2003*, pages 294–311. Springer, 2003.

[53] Rachna Dhamija and J Doug Tygar. The battle against phishing: Dynamic security skins. In *Proceedings of the 2005 symposium on Usable privacy and security*, pages 77–88. ACM, 2005.

[54] B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo. *Baiting Inside Attackers Using Decoy Documents*, pages 51–70. 2009.

[55] B. M. Bowen, V. P. Kemerlis, P. Prabhu, A. D. Keromytis, and S. J. Stolfo. Automating the injection of believable decoys to detect snooping. In *WiSec*, pages 81–86. ACM, 2010.

[56] A. Juels and R. Rivest. Honeywords: Making password-cracking detectable. In *ACM Conference on Computer and Communications Security – CCS 2013*, pages 145–160. ACM, 2013.

[57] A. Juels. A bodyguard of lies: the use of honey objects in information security. In *SACMAT*, pages 1–4, 2014.

[58] John Kelsey, Bruce Schneier, Chris Hall, and David Wagner. Secure applications of low-entropy keys. In *Information Security*, pages 121–134. Springer, 1998.

[59] Xavier Boyen. Halting Password Puzzles – Hard-to-break Encryption from Human-memorable Keys. In *16th USENIX Security Symposium*, pages 119–134. Berkeley: The USENIX Association, 2007. Available at http://www.cs.stanford.edu/~xb/security07/.

[60] Eran Gabber, Phillip B. Gibbons, Yossi Matias, and Alain J. Mayer. How to Make Personalized Web Browising Simple, Secure, and Anonymous. In *FC '97: Proceedings of the 1st International Conference on Financial Cryptography*, pages 17–32, London, UK, 1997. Springer-Verlag.

[61] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J.C. Mitchell. Stronger password authentication using browser extensions. In *USENIX Security*, 2005.

[62] J. Alex Halderman, Brent Waters, and Edward W. Felten. A Convenient Method for Securely Managing Passwords. In *WWW '05: Proceedings of the 14th International Conference on World Wide Web*, pages 471–479, New York, NY, USA, 2005. ACM.

[63] Sonia Chiasson, P.C. van Oorschot, and Robert Biddle. A Usability Study and Critique of Two Password Managers. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[64] Ka-Ping Yee and Kragen Sitaker. Passpet: convenient password management and phishing protection. In *Proceedings of the second symposium on Usable privacy and security*, pages 32–43. ACM, 2006.

[65] Daniel McCarney, David Barrera, Jeremy Clark, Sonia Chiasson, and Paul C. van Oorschot. Tapas: Design, Implementation, and Usability Evaluation of a Password Manager. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 89–98, New York, NY, USA, 2012. ACM.

[66] Ambarish Karole, Nitesh Saxena, and Nicolas Christin. A comparative usability evaluation of traditional password managers. In Kyung-Hyune Rhee

and DaeHun Nyang, editors, *Information Security and Cryptology - ICISC 2010*, volume 6829 of *Lecture Notes in Computer Science*, pages 233–251. Springer Berlin Heidelberg.

[67] Dinei Florencio and Cormac Herley. A large-scale study of web password habits. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 657–666, New York, NY, USA, 2007. ACM.

[68] Blase Ur, Fumiko Noma, Jonathan Bees, Sean M Segreti, Richard Shay, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. "I added '!' at the end to make it secure": Observing password creation in the lab. In *Eleventh Symposium On Usable Privacy and Security (SOUPS 2015)*, pages 123–140, 2015.

[69] Blase Ur, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas, Lujo Bauer, et al. How does your password measure up? the effect of strength meters on password creation. In *USENIX Security Symposium*, pages 65–80, 2012.

[70] Y. Zhang, F. Monrose, and M. K. Reiter. The security of modern password expiration: an algorithmic framework and empirical analysis. In *ACM Conference on Computer and Communications Security (ACM CCS)*, pages 176–186, 2010.

[71] Richard Shay, Patrick Gage Kelley, Saranga Komanduri, Michelle L Mazurek, Blase Ur, Timothy Vidas, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Correct horse battery staple: Exploring the usability of system-assigned passphrases. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, page 7. ACM, 2012.

[72] Richard Shay, Saranga Komanduri, Adam L Durity, Phillip Seyoung Huh, Michelle L Mazurek, Sean M Segreti, Blase Ur, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Can long passwords be secure and usable? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2927–2936. ACM, 2014.

[73] Joseph Bonneau and Stuart Schechter. Towards reliable storage of 56-bit secrets in human memory. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX, 2014.

[74] Mark Keith, Benjamin Shao, and Paul John Steinbart. The usability of

passphrases for authentication: An empirical field study. *International journal of human-computer studies*, 65(1):17–28, 2007.

[75] Mark Keith, Benjamin Shao, and Paul Steinbart. A behavioral analysis of passphrase design and effectiveness. *Journal of the Association for Information Systems*, 10(2):2, 2009.

[76] Alec Muffet. Facebook: Password hashing & authentication. Presentation at Real World Crypto, 2015.

[77] Emil Protalinski. Facebook passwords are not case sensitive. http://www.zdnet.com/article/facebook-passwords-are-not-case-sensitive-update/, 2011. Accessed: 2015-11-12.

[78] S. Antilla. Vanguard group fires whistleblower who told thestreet about flaws in customer security., 2015.

[79] Is Vanguard making it too easy for cybercriminals to access your account? http://www.thestreet.com/story/13213265/4/is-vanguard-making-it-too-easy-for-cybercriminals-to-access-your-account.html?startIndex=0. Accessed: 2015-11-06.

[80] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, Thomas Ristenpart, and Cornell Tech. The Pythia PRF service. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 547–562. USENIX Association, 2015.

[81] M. Siegler. One of the 32 million with a RockYou account? you may want to change all your passwords. like now. *TechCrunch*, 14 Dec. 2009.

[82] Michelle L Mazurek, Saranga Komanduri, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Patrick Gage Kelley, Richard Shay, and Blase Ur. Measuring password guessability for an entire university. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 173–186. ACM, 2013.

[83] Y. Dodis, L. Reyzin, and A. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In C. Cachin and J. Camenisch, editors, *Eurocrypt 2004*, pages 523–540. Springer-Verlag, 2004. LNCS no. 3027.

[84] Andrew Mehler and Steven Skiena. Improving usability through password-

corrective hashing. In *String Processing and Information Retrieval*, pages 193–204. Springer, 2006.

[85] Randall Munroe. Password strength. https://xkcd.com/936/, 2015. Accessed: 2015-11-13.

[86] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[87] Fred J Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[88] Gregory V Bard. Spelling-error tolerant, order-independent pass-phrases via the Damerau-Levenshtein string-edit distance metric. In *Proceedings of the fifth Australasian Symposium on ACSW Frontiers-Volume 68*, pages 117–124. Australian Computer Society, Inc., 2007.

[89] Markus Jakobsson and Ruj Akavipat. Rethinking passwords to adapt to constrained keyboards. *Proc. IEEE MoST*, 2012.

[90] Amazon login may accept password variants. http://www.ghacks.net/2011/01/31/amazon-login-may-accept-password-variants/. Accessed: 2015-11-06.

[91] Michael Buhrmester, Tracy Kwang, and Samuel D Gosling. Amazon's mechanical turk a new source of inexpensive, yet high-quality, data? *Perspectives on psychological science*, 6(1):3–5, 2011.

[92] Sascha Fahl, Marian Harbach, Yasemin Acar, and Matthew Smith. On the ecological validity of a password study. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, page 13. ACM, 2013.

[93] Uniq Turker. https://uniqueturker.myleott.com/. Accessed: 2015-15-10.

[94] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[95] Dan Wheeler. zxcvbn: Realistic password strength estimation. Dropbox Tech Blog, April 2012. https://blogs.dropbox.com/tech/2012/04/zxcvbn-realistic-password-strength-estimation/.

[96] Colin Percival and Simon Josefsson. The scrypt password-based key derivation function. 2015.

[97] Alex Biryukov, D Dinu, and D Khovratovich. Argon and argon2: password hashing scheme. Technical report, Technical report, 2015.

[98] S Boztas. Entropies, guessing, and cryptography. *Department of Mathematics, Royal Melbourne Institute of Technology, Tech. Rep*, 6:2–3, 1999.

[99] Benjamin Fuller, Adam Smith, and Leonid Reyzin. When are fuzzy extractors possible? Cryptology ePrint Archive, Report 2014/961, 2014. https://eprint.iacr.org/2014/961.pdf.

[100] Dorit S Hochbaum. Approximating covering and packing problems: set cover, vertex cover, independent set, and related problems. In *Approximation algorithms for NP-hard problems*, pages 94–143. PWS Publishing Co., 1996.

[101] Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.

[102] Twitter's list of 370 banned passwords. http://www.businessinsider.com/twitters-list-of-370-banned-passwords-2009-12. Accessed: 2015-11-06.

[103] Dan Lowe Wheeler. zxcvbn: Low-budget password strength estimation. In *Proc. USENIX Security*, 2016.

[104] William Melicher, Blase Ur, Sean M Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. Fast, lean and accurate: Modeling password guessability using neural networks.

[105] Michel Abdalla, Mihir Bellare, and Gregory Neven. Robust encryption. *Journal of Cryptology*, pages 1–44, 2010.

[106] Payman Mohassel. A closer look at anonymity and robustness in encryption schemes. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 501–518. Springer, 2010.

[107] Pooya Farshim, Benoît Libert, Kenneth G Paterson, Elizabeth A Quaglia, et al. Robust encryption, revisited. In *Public Key Cryptography*, volume 7778, pages 352–368. Springer, 2013.

[108] Pooya Farshim, Claudio Orlandi, and Razvan Rosie. Security of symmetric primitives under incorrect usage of keys. *IACR Transactions on Symmetric Cryptology*, 2017(1):449–473, 2017.

[109] Saranga Komanduri, Richard Shay, Patrick Gage Kelley, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Serge Egelman. Of passwords and people: measuring the effect of password-composition policies. In *CHI*, 2011.

[110] Frank Wilcoxon, SK Katti, and Roberta A Wilcox. *Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test*. American Cyanamid Company, 1963.

[111] Vipin Samar. Unified login with pluggable authentication modules (pam). In *Proceedings of the 3rd ACM conference on Computer and communications security*, pages 1–10. ACM, 1996.

[112] Kenneth Raeburn. Advanced encryption standard (aes) encryption for kerberos 5. 2005.

[113] Patrick Biernacki and Dan Waldorf. Snowball sampling: Problems and techniques of chain referral sampling. *Sociological methods & research*, 10(2):141–163, 1981.

[114] List of data breaches. https://en.wikipedia.org/wiki/List_of_data_breaches, 2018.

[115] Troy Hunt. Have I Been Pwned? https://haveibeenpwned.com/Passwords/, 2018.

[116] Testing Firefox Monitor, a new security tool. https://blog.mozilla.org/futurereleases/2018/06/25/testing-firefox-monitor-a-new-security-tool/, 2019.

[117] Security update - q2 2018. https://www.eveonline.com/article/pc29kq/an-update-on-security-the-fight-against-bots-and-rmt, 2018.

[118] Finding Pwned Passwords with 1Password. https://blog.agilebits.com/2018/02/22/finding-pwned-passwords-with-1password/, 2018.

[119] Ágnes Kiss, Jian Liu, Thomas Schneider, N Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. *Proceedings on Privacy Enhancing Technologies*, 2017(4):177–197, 2017.

[120] Bijeeta Pal, Tal Daniel, Rahul Chatterjee, and Thomas Ristenpart. Beyond credential stuffing: Password similarity using neural networks. *IEEE Symposium on Security and Privacy*, may 2019.

[121] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *NDSS*, volume 14, pages 23–26, 2014.

[122] Ding Wang, Zijian Zhang, Ping Wang, Jeff Yan, and Xinyi Huang. Targeted online password guessing: An underestimated threat. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1242–1254. ACM, 2016.

[123] Jennifer Pullman, Kurt Thomas, and Elie Bursztein. Protect your accounts from data breaches with Password Checkup. https://security.googleblog.com/2019/02/protect-your-accounts-from-data.html, 2019.

[124] Julio Casal. 1.4 billion clear text credentials discovered in a single database. https://medium.com/4iqdelvedeep/1-4-billion-clear-text-credentials-discovered-in-a-single-database-3131d0a1ae14, Dec, 2017.

[125] I wanna go fast: Why searching through 500m pwned passwords is so quick. https://www.troyhunt.com/i-wanna-go-fast-why-searching-through-500m-pwned-passwords-is-so-quick/, 2018.

[126] Steven Englehardt, Jeffrey Han, and Arvind Narayanan. I never signed up for this! privacy implications of email tracking. *Proceedings on Privacy Enhancing Technologies*, 2018(1):109–126, 2018.

[127] Tor (anonymity network). https://en.wikipedia.org/wiki/Tor_(anonymity_network).

[128] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious prf with applications to private set intersection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 818–829. ACM, 2016.

[129] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 515–530, 2015.

[130] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based psi via cuckoo hashing. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 125–157. Springer, 2018.

[131] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1243–1255. ACM, 2017.

[132] Yue Li, Haining Wang, and Kun Sun. A study of personal information in human-chosen passwords and its security implications. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[133] P. Berenbrink, T. Friedetzky, Z. Hu, and R. Martin. On weighted balls-into-bins games. *Theoretical Computer Science*, 409(3):511–520, 2008.

[134] Junade Ali. Validating Leaked Passwords with k-Anonymity. https://blog.cloudflare.com/validating-leaked-passwords-with-k-anonymity/, 2018.

[135] A Naranyanan and V Shmatikov. Robust de-anonymization of large datasets. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy, May 2008*, 2008.

[136] Ashwin Machanavajjhala, Johannes Gehrke, Daniel Kifer, and Muthuramakrishnan Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 24–24. IEEE, 2006.

[137] Lei Zhang, Sushil Jajodia, and Alexander Brodsky. Information disclosure under realistic assumptions: Privacy versus optimality. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 573–583. ACM, 2007.

[138] Junade Ali. Optimising caching on pwned passwords (with workers). https://blog.cloudflare.com/optimising-caching-on-pwnedpasswords, 2018.

[139] Interval tree. https://en.wikipedia.org/wiki/Interval_tree, 2018.

[140] Argon2. https://www.npmjs.com/package/argon2/, 2018.

[141] Dynamodb. https://aws.amazon.com/dynamodb/, 2018.

[142] Cloudfront. https://aws.amazon.com/cloudfront/, 2018.

[143] Crypto nodejs. https://nodejs.org/api/crypto.html, 2019.

[144] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[145] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. Xpir: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies*, 2016(2):155–174, 2016.

[146] Femi Olumofin and Ian Goldberg. Revisiting the computational practicality of private information retrieval. In *International Conference on Financial Cryptography and Data Security*, pages 158–172. Springer, 2011.

[147] Zeev Dvir and Sivakanth Gopi. 2-server pir with sub-polynomial communication. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 577–584. ACM, 2015.

[148] Vericlouds. https://my.vericlouds.com/, 2019.

[149] GhostProject. https://ghostproject.fr/, 2019.

[150] Joanne Woodage, Rahul Chatterjee, Yevgeniy Dodis, Ari Juels, and Thomas Ristenpart. A new distribution-sensitive secure sketch and popularity-proportional hashing. In *Annual International Cryptology Conference*, pages 682–710. Springer, 2017.

[151] Marie-Sarah Lacharité and Kenneth G Paterson. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. *IACR Transactions on Symmetric Cryptology*, 2018(1):277–313, 2018.

[152] Salvatore Aurigemma, Thomas Mattson, and Lori Leonard. So much

promise, so little use: What is stopping home end-users from using password manager applications? 2017.

[153] Wikitionary: Frequency List. http://en.wiktionary.org/wiki/Wiktionary:Frequency_lists.

[154] Ron Bowes. Skull Security, Passwords. https://wiki.skullsecurity.org/Passwords.

[155] Tetsu Fujisaki, Fred Jelinek, John Cocke, Ezra Black, and T Nishino. A probabilistic parsing method for sentence disambiguation. In *Current Issues in Parsing Technology*, pages 139–152. Springer, 1991.

[156] Y. Dodis. On extractors, error-correction and hiding all partial information. In *Theory and Practice in Information-Theoretic Security, 2005. IEEE Information Theory Workshop on*, pages 74–79, 2005.

[157] Rafail Ostrovsky and Yuval Rabani. Low distortion embeddings for edit distance. *Journal of the ACM (JACM)*, 54(5):23, 2007.

[158] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. An analysis of approximations for maximizing submodular set functions—i. *Mathematical Programming*, 14(1):265–294, 1978.

[159] Andreas W Hauser and Klaus U Schulz. Unsupervised learning of edit distance weights for retrieving historical spelling variations. In *Proceedings of the First Workshop on Finite-State Techniques and Approximate Search*, pages 1–6, 2007.

[160] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. In *Advances in Cryptology–EUROCRYPT*, volume 4004, page 10, 2006.

[161] Godfrey Harold Hardy, John Edensor Littlewood, and George Pólya. *Inequalities*. Cambridge university press, 1952.