HardPass: A Keyboard-Centric Password Manager

Lev Grossman, Varun Jain, Michael Wornow
Harvard University
{lgrossman, jainv, mwornow}@college.harvard.edu

***Abstract*—Just as a chain is often said to be only as strong as its weakest link, so too are security systems. The overly simplistic and generally predictable nature of most passwords presents a serious threat in today's digitized world, when all of one's private information – social security number, credit card and banking information, home address, etc. – can be accessed with a simple password. We therefore propose a novel solution to the challenge of password management: HardPass. HardPass works by creating a unique per-user keyboard layout that maps input key-strokes to randomized output character sequences. In this way, even the simplest of passwords (e.g. "kitties") can become an extremely complex, high entropy password, safe from even the most state-of-the-art brute force password crackers. In this paper we will compare HardPass with more conventional password managers, detail the security and usability performance of HardPass, and finally discuss areas for future work.**

Keywords*—*Password Manager, Keyboard Layout, Password Entropy

I. Introduction

Passwords – namely simple, text-based ones – are by far the most popular method of user verification online [5]. Used by essentially every website on the internet offering individual user accounts, text-based passwords are virtually synonymous with the web – they are everywhere. Even offline these password systems dominate the user authentication space. From securing personal computer accounts to desktop-based applications, text-based passwords are king.

Despite their ubiquity, however, passwords are riddled with security vulnerabilities [2], especially because most users often trade password entropy for memorability. While memorizing extremely long, pattern-less passwords would help to ensure their digital safety, most people do not go to sufficient lengths to ensure that their account passwords are as secure as possible. Rather, it is much easier for a user to simply choose a password that is memorable (and thus typically lower entropy). This is dangerous, for the passwords users choose are often either very common or are composed of words or names, making them susceptible to dictionary-based attacks [6]. Additionally, because a single user can have dozens of unique internet accounts, it is often easier to choose one single password – and perhaps vary it slightly across accounts – instead of multiple unique ones. This is problematic, for if a user's password is stolen for one account then an attacker could potentially use that password to break into that user's other accounts.

While most sites recommend that users select highly random, long passwords, this advice is often not heeded. Indeed, when sites actually try to enforce stricter password guidelines, many users find it overwhelmingly frustrating to remember and use these "more secure" passwords [3]. In addition to increased user frustration, these "security recommendations" are often not much more effective in ensuring that users pick higher entropy passwords [5]. Oftentimes when sites require users to include an uppercase letter, special symbol, or number in their password, users simply insert these characters at the beginning or the end of their password. This predictability, in turn, helps keep the entropy of user passwords low, helping state-of-the-art password cracking systems easily brute force even these supposedly more secure passwords.

Although passwords have plenty of problems they have continued to dominate the security space due to the drawbacks of proposed alternatives [4]. One of the most common drawbacks of these other security models is their lack of usability. Bonneau et al. goes into great detail explaining the benefits and drawbacks of different authentication schemes [1]. The authors conclude that text-based passwords, while sub-optimal in many respects (namely security), actually stack up quite favorably compared to possible alternatives when taking a holistic view of user authentication systems (i.e. take into account usability, deployability, and security). When later analyzing the performance of HardPass we will subject the system to the same holistic grading as done in [1] to rigorously assess its merits relative to other potential user authentication models.

Because text-based passwords are so popular – and unlikely to be replaced by other authentication schemes – many of the more successful recent efforts towards improving web security have attempted to create more secure or user-friendly systems that supplement rather than replace vanilla passwords: password managers, two-factor authenticators, etc. We will discuss, in depth, a representative selection of these solutions in the following section. While all the proposed schemes discussed in Bonneau et al. succeeded in some categories, most had either large usability or security vulnerabilities – generally, the more usable, the less secure, and vice versa. Thus, in this paper we propose and discuss HardPass[1] as a syste that is both much more secure than normal text-based passwords and equally as user-friendly. Moreover, HardPass can be seamlessly used in conjunction with these supplemental authentication schemes (i.e. two factor authentication), thereby improving the security of the weakest link in any computer-based authentication system: low entropy passwords.

A. Paper Organization

In the remainder of this paper we will describe our approach for a better password system that is both highly secure and user friendly. In Section 2 we will discuss alternate password systems and their shortcomings. In Section 3 we will explain the theory and motivation behind our keyboard

[1] HardPass Source: https://github.com/grossmanlev/CS263-final-project

randomization approach. In Section 4 we will discuss keyboard driver technology across various platforms and how we handled the driver implementation of HardPass. In Section 5 we will introduce our cloud-based HardPass implementation, which utilizes a Python web server and Chrome browser extension. In Section 6, we will analyze the performance of the entire HardPass system – both server and drivers. And finally, in Sections 7 and 8 we will draw conclusions and discuss future work.

II. Related Work

In this section, we will present and discuss the benefits and drawbacks of some alternatives to simple text-based password schemes. While we do not claim to cover all alternatives here, we do believe the following selection is representative of the most popular solutions in the field.

A. Password Managers

The need for users to remember dozens of different passwords for dozens of different services is one of the key reasons why passwords are often short and simple to crack. They are limited not by security systems themselves but by the limits of human memory. Password managers help address this problem by storing all of a user's many passwords in one centralized location, and auto-filling password fields with those stored passwords. Users must remember only one, hopefully high entropy, password to their password manager system. In addition to being incredibly convenient for users with many different accounts, password managers can also strengthen a users' passwords by choosing on their behalf a long, randomly generated string for each of that user's accounts. Thus, a user that has a password manager like LastPass or Keeper does not need to create a password for a new website, let alone remember it, for the password manager takes care of generating a high-entropy password and storing it on the user's behalf.

Though current commercial password managers vary greatly in terms of their exact deployment, the general model for password managers that this paper will analyze will be based on that of LastPass. LastPass is the leading commercial password manager currently on the market, with over 16.5 million individual users and 43,000 enterprise customers [16]. It is a cloud-based password manager, meaning that a user's encrypted "vault" of account passwords being managed by LastPass is stored on a database in the cloud. This is in contrast to local-based password managers like KeePass, which only stores passwords on a user's local computer. Though a local-only password manager has a much smaller threat surface than a cloud-based one like LastPass, the obvious drawback is that a user with multiple devices will have a difficult time syncing their locally-managed passwords across their devices. Especially for non-technical users (the ones most in need of a password manager in the first place), this inconvenience makes a solution like KeePass infeasible. Thus, in addition to security, one of the most important considerations for any password manager service is usability and multi-device support, which helps to explain why LastPass is consistently rated the "#1 password manager." Thus, we will use LastPass's robust and market-leading cloud-based model of password management as a baseline to compare our new service, HardPass.

Though LastPass is not extremely forthcoming about its infrastructural details, we were able to piece together a high-level overview of its security architecture based on technical whitepapers, reporting on previous hacks and vulnerabilities, and analysis by Harvard Professor of Computer Science, James Mickens. When a user creates an account with LastPass, his password (the "Master Password") is first hashed (with his username as the salt) on the client-side using 100,100 rounds of PBKDF2-SHA256. Two outputs of this hashing process are generated. The first output is hashed one more time to generate the "Authentication Hash", which is sent to the LastPass server and will be used to verify the user the next time he logs in. The second output is the "Encryption Key" which will be used as the key to a 256-bit AES function that will encrypt the "Vault" of passwords that the user stores with LastPass. The passwords stored in the "Vault" are passwords for sites like Facebook, Google, Twitter, etc. As part of its marketing, LastPass stresses that it never has access to your Master Password or Encryption Key – "Your master password, and the keys used to encrypt and decrypt data, are never sent to LastPass servers, and are never accessible by LastPass" – and that it uses "AES-256 bit encryption with PBKDF2 SHA-256 and salted hashes to ensure complete security in the cloud" [17].

In order to support multi-device usage, however, LastPass must store the encrypted Vault of passwords on its cloud server. Thus, both the Authentication Key of the Master Password and the encrypted Vault of account passwords are stored on the LastPass server. This architecture is inherent to every cloud-based password manager (and thus every commercially successful password manager), for barring complex peer-to-peer communication protocols or physical data transfers there is no simple, quick, and economical method for synchronizing passwords across multiple devices. This, however, is the core weakness of any cloud-based password manager, for if an attacker is able to penetrate LastPass's servers and gain access to their content, then the attacker will be able to steal the encrypted Vault and Authentication Keys and run brute force attacks against them. And if the attacker is successful then he will be able to log in to every account that this unlucky victim has trusted LastPass to safely store.

In practice, LastPass stores its Authentication Keys on one physical server and its encrypted Vaults on another physical server. The benefits of this extra layer of isolation became very clear in 2015, when attackers were able to hack into the Authentication Key server but not the Vault server [18]. Thus, hackers were able to steal users' LastPass usernames (i.e. the salt for the Master Password hashing process) as well as their "Final Authentication Hash", which could be used to decrypt these users' Vault of passwords. Hackers were not, however, able to penetrate the other server containing the users' password Vaults. This limited the damage that attackers could do to just brute-forcing user passwords and logging in to LastPass. Because of the PBKDF2-SHA256 hashing used by LastPass, however, this would take a significant amount of time and resources to accomplish. Thus, damage was mitigated by LastPass by forcing users to reset their passwords [19].

Had attackers also been able to penetrate the other LastPass server containing the encrypted Vaults, however, then a true catastrophe would have ensued. This is because the attacker could then simply use the stolen Final Authentication Hashes

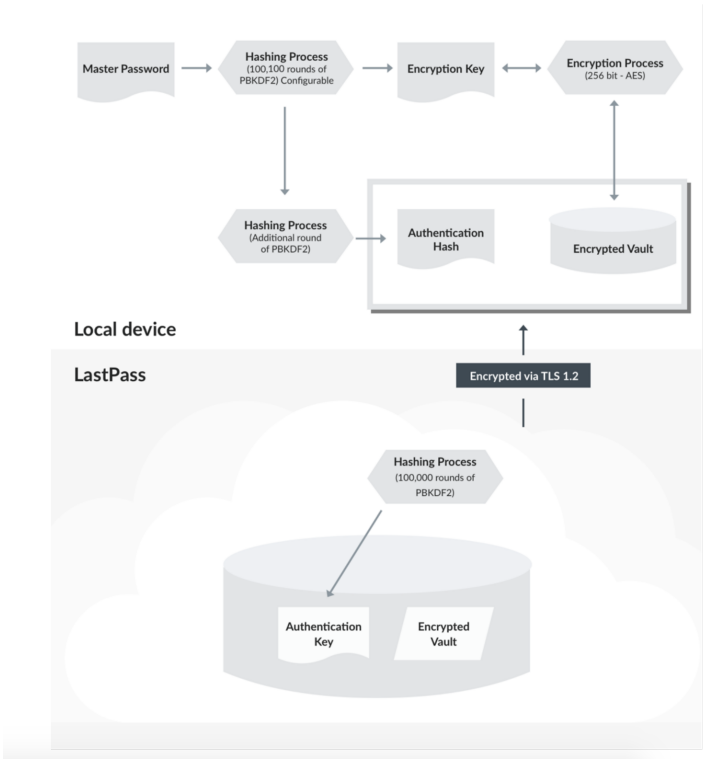


Fig. 1: High-level diagram of LastPass's security architecture

to decrypt each encrypted Vault, read the plain text passwords and corresponding web services stored in each user's Vault, and then log in to every account that the user had trusted LastPass to store his password for. At that point there would have been nothing LastPass could do, for just resetting one's LastPass password would have been too late – every user would have had to also reset his Facebook, Google, Twitter, etc. passwords before the attackers were able to log in to their accounts. It was only through sheer luck that LastPass did not find itself in this predicament [19].

Thus, the general issue with a cloud-based password manager like LastPass is that it must still store a users' passwords somewhere, i.e. the cloud. Yes, the randomized passwords that LastPass chooses for its users' Facebook, Google, Twitter, etc. accounts have much higher entropy than the passwords that users would otherwise have chosen. However, if the password manager's centralized database of passwords is compromised, an attacker would still be able to log in to every users' accounts regardless of how high-entropy their passwords are – it doesn't matter whether a password is `password123` or `(VjJ‘9jNW0-!3mwk.[9]<q` if an attacker can see its plaintext anyway.

HardPass seeks to address this issue by **never storing any user passwords**. As is detailed elsewhere in the paper, by only storing key mappings and not actual passwords, HardPass essentially has a "built-in" 2-factor authentication – the second-factor being the user actually typing in his password. A complete compromise of HardPass's servers would not reveal any passwords, even if the attacker were able to decrypt the contents of HardPass's servers.

B. 2-Factor Authentication

Two-factor authentication schemes seek to provide an extra layer of security protection by confirming one's identity using not only something they know (e.g. passwords), but also either something they have (e.g. mobile phone) or something they are (e.g. biometrics) [27]. In the simplest implementations, a temporary code may be texted to the mobile device of a user, once a password has been correctly entered. Other two-factor schemes use third-party apps, such as Duo Mobile or Google Authenticator, which can provide additional functionality. These include secure one-time password generation schemes, the ability to work without network connection, and even useful analytics for an organization. In addition to third-party apps, hardware-based two-factor authentication schemes have gained prominence, as they can be more resistant to tampering. For example, Yubikeys [28] provide for hardware-based one-time password generation, without requiring any transcription on the part of the user. Despite the security benefits provided by two-factor authentication schemes, adoption of such schemes is unfortunately limited [29] as users are often required to carry an auxiliary device.

C. Non Text-Based Passwords

Despite our focus on text-based and text-centric password schemes, we will briefly mention a few non text-based systems.

1) Graphical Passwords: Instead of focusing on a user's ability to remember a piece of text, graphical passwords rely on a user's ability to remember images. A good example of this type of password scheme is the Persuasive Cued Click-Point (PCCP) system [20]. The PCCP "sign-in" process requires a

user to pick a specific point in each of five unique images. Then, upon log-in, the user needs to pick the same five points in the same five images – within a particular tolerance range. The benefits of the system are that it eliminates text-based brute force attacks entirely, and makes graphical brute force attacks difficult – as the points in the images are chosen at random. Indeed, security analysis of the system concludes PCCP offers competitive security guarantees [21]. However, due to the text-less nature of the system, it requires special integration and actually incurs longer log-in times than text-based password schemes.

2) Biometrics: Just like in Graphical Passwords, Biometrics allows an authentication scheme to avoid having users type in text-based passwords entirely. Instead, Biometrics verifies users based on a variety of biological features: voice, eyes, fingerprints, etc [22]. With the advent of cheaper hardware solutions, fingerprint verification schemes have become increasingly popular, especially in the mobile phone space [23]. Despite fingerprint scanning being quite usable, there's been a lot of work done on tricking fingerprint scanners [24], and implementing these systems causes a deployability issue – due to specific implementation being necessary.

3) Zero-Knowledge Passwords: Similar in nature to Zero-Knowledge Proofs [25], these password schemes attempt to provide user authentication without having the user actually divulge their "password" to the authentication server. A good example of this type of system is the GrIDsure scheme [26]. Upon sign-in, a user is prompted with a 5×5 grid and required to choose a length-4 pattern. Then, whenever a user logs in, they are prompted with the same 5×5 grid, this time populated with numbers. In order to successfully log in, the user needs to input the numbers corresponding to their 4-pattern – this way, the authentication server doesn't actually get their password on every log-in attempt. Although this helps avoid replay attacks, it again is not easily deployable, and the log-in process takes longer than traditional text-based methods.

III. Randomized Keyboard

As the title of the paper implies, the key idea behind our password scheme is the randomized keyboard. Specifically, we define a randomized keyboard to be a one that maps inputted key presses to randomized outputs. We will discuss the implementation of these keyboards in future sections, but for now, we will discuss the theoretical benefits of these randomized keyboards in a password scheme setting.

One of the biggest problems with passwords, as previously mentioned, is the lack of entropy – users often trade password security for memorability. If a user could successfully remember a highly random, long password, many of the security drawbacks of simple text-based passwords would disappear (e.g. brute force approaches would be rendered virtually meaningless). How, then, could a user have a high entropy password without actually being required to memorize that password?

Our solution to this problem is to create for each user a unique, randomized keyboard. With the help of this randomization, users can enjoy all the benefits of using a high entropy password without the drawbacks of having to actually remember that password. Note that our current implementation of the non-web-based schemes randomizes only the alphanumeric keys (and their shift modifiers). However, it is easy to extend the system to also randomize the comma key, period key, tilda key, etc.

A. Keyboard Usage

When designing the HardPass system, we wanted to focus on usability. Therefore, it is easy to switch to the HardPass randomized keyboard whenever a user chooses (Figure 3). We will go into more depth in future sections as to the system-specific details of keyboard switching, but generally switching keyboards is as easy as switching your keyboard input language – which can be accomplished on most systems via a system-specific GUI or the press of a hot-key. In fact, as we will explain in the HardPass Web-server section, our Google Chrome plugin actually does keyboard switching automatically. Therefore, when a user is required to enter a password, they can easily switch to their HardPass keyboard – or wait for it to switch automatically – and simply type in as normal. Their keystrokes will automatically be converted into the randomized output defined by their particular keyboard, and outputted to the password field – or whatever other environment in which they are typing.

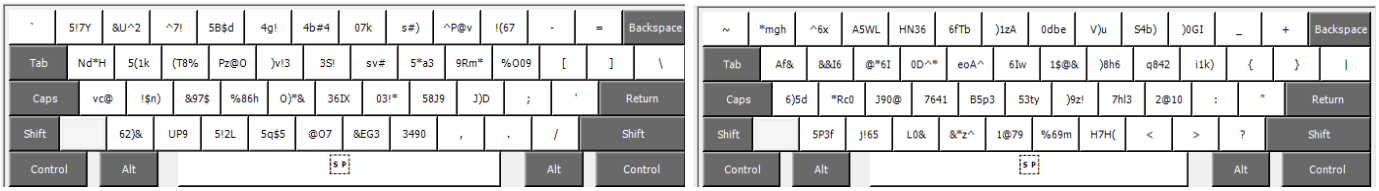
B. Randomization

For the HardPass system, we experimented with a couple different randomization variants – simple key-to-key randomization, and a key-to-string randomization. We will discuss both is depth.

The simpler of the two variants is the key-to-key randomization. Namely, we permute the keyboard such that any given key input will map to a random key output. For example, a particular HardPass key-to-key keyboard could deterministically map the key `a` to `o`, `&` to `2`, etc. This particular keyboard scheme would help increase the entropy of passwords, especially for those that contain English words, names, or other easily recognizable strings. As previously mentioned, one of the biggest problems with passwords is that users often put meaningful words in their passwords. While easy to remember, these strings make it easy for attackers to use dictionary-based attacks to easily crack passwords [6]. With this key-to-key randomization, there is a high probability that passwords containing English words would be randomized to complete gibberish – not susceptible to dictionary-based attacks.

Despite the key-to-key scheme being able to obfuscate passwords, they do not add extra protection through length. If a user password is fewer than 4 or 5 characters long, their randomized password will still be short – and thus possibly susceptible to simple brute-force approaches. To solve for this, we have also implemented a key-to-string randomized keyboard. This particular keyboard maps keys to random strings of 3 or 4 characters. For example, `a` may map to `4%hY`, `&` to `B7q!`, etc. This ensures that even short user passwords are mapped to longer passwords – namely one that is on average 3 to 4 times longer than the original. This added length greatly helps increase overall password entropy.

In addition, when originally coming up with these randomized mappings – specifically for the non-web-based scheme – we categorize the keyboard into three groups: alphabetic



(a) Without Shift

(b) With Shift

Fig. 2: HardPass keyboard – without and with the shift modifier

(A), numeric (N), and special (S). The alphabetic group contains all lowercase and uppercase letters of the English alphabet: {a, ..., z, A, ..., Z}. The numeric group contains all the numbers, {0, ..., 9}, and the special group currently contains all the shift modifiers of the number keys: {!, @, #, $, %, ^, &, *, (,)}. Although OWASP's official list of special symbols defines a larger group of characters, we took a more conservative stance here due to restrictions some sites place on acceptable special symbols [9].

Using these three categories, we define the randomized string mapping s_k to each key k using the following procedure:

1) Set the length of s_k according to the following probabilities: $p(|s_k| = 3) = 0.25$ and $p(|s_k| = 4) = 0.75$.
2) For $i \in \{1, ..., |s_k|\}$, select a group, G_i at random from $\{A, N, S\}$, and set the ith character, $s_k^{(i)}$ to a random character, c from G_i.

C. Implementation

In implementing the HardPass randomized keyboard system, we decided to split the scheme into two major components: a system-level component and a web component. The system-level component would mainly consist of downloadable custom keyboard drivers, while the web component would consist of a fully integrated HardPass web app – modeled after the currently popular password manager applications. In the following two sections, we will go into greater depth about the specifics of the two types of systems implemented.

IV. Keyboard Drivers

The first of the two systems we implemented is the system-specific randomized keyboard driver. These drivers can be downloaded straight to a user's computer, and after installation enable a user to switch to their unique HardPass keyboard in the same manner that they'd switch to an international keyboard on their machine. We will now discuss the OS-specific details of the drivers.

A. Windows

For the purpose of this section, we will discuss the HardPass implementation on a 64-bit machine running Windows 10. However, most of the details would be similar for 32-bit systems as well as for previous versions of Windows – XP, Vista, 7, and 8.

On Windows 10 (as well as previous versions of Windows), keyboard layouts can be specified with the special `.klc` file format – the file simply specifies keyboard mappings as well as any other special modifications. Despite poor documentation of the `.klc` file format, we were able to reverse-engineer the pertinent aspects of the file using Microsoft's Keyboard Layout Creator [13]. This graphical application allows for users to create custom keyboard layouts and install those layouts on their machine. In order to make the HardPass installation process simple, we reverse-engineered most of the MKLC program in order to automate the creation and installation of the randomized keyboard.

As mentioned, the first step in creating a custom keyboard layout is writing the `.klc` file. Due to a Windows-specific limitation of 4 characters per key mapping, we chose to implement the key-to-string keyboard randomization – as described in the Randomized Keyboard section. A script we wrote automatically creates this new randomized HardPass keyboard layout and saves it in the `.klc` file format. Once this is done, the next step is to actually install the new keyboard layout on the machine.

On Windows, keyboard drivers take the form of dynamically linked library (DLL) files stored in the `C:\Windows\System32` and `C:\Windows\WOW64` directories [14]. Ironically, the 32-bit version of the keyboard driver is stored in `WOW64` and the 64-bit library is stored in `System32`. In addition to the two DLL's, installing a custom keyboard driver requires the addition of a custom registry file in `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Keyboard Layouts\` – this registry directory contains the specifications for all keyboards installed on the computer.

In order to complete this installation step, we wrote a script that first compiles the HardPass `.klc` keyboard layout file into two DLL's – 32 and 64-bit respectively – using the MKLC `.klc` compiler. After moving these libraries to the correct directories, we then import the custom registry file into the Windows Registry. A simple system restart is then all that is necessary for the new HardPass keyboard to appear in the user's list of keyboards.

B. Linux

We implement a version of the keyboard randomization software for Ubuntu 18.04. Keyboard layout configurations are managed through X.org's xkb software [15]. Specifically, under the `/usr/share/X11/xkb/symbols/` directory, one can find the specifications for the keyboard layouts of various languages. To a first approximation, each of these layout files–`us`, for example–contains a mapping from a key

to four symbols. These symbols specify the output associated for each key when it is pressed unmodified, with shift, with the right alt key, and with shift + right alt.

Our implementation randomly maps unmodified keys and shifted keys to symbols. It creates the appropriate file with these mappings in the `/usr/share/X11/xkb/symbols/` directory; note that this step requires administrative privileges. In order to switch from the normal keyboard layout to the HardPass keyboard layout and vice versa, the user can use keyboard shortcuts Ctrl + $\rightarrow$ and Ctrl + $\leftarrow$, respectively. Under the hood, these shortcuts use the `setxkbmap` command to set the keyboard layout configuration.

Currently, the Ubuntu implementation maps each key to only 1 random symbol. Compared to the Windows implementation, such a system is less secure, as there are fewer bits of randomness. In terms of usability, backspacing is more intuitive for the user; one backspace undoes one key stroke. This is not the case for the Windows implementation. On the other hand, with only 1 random symbol per key, it is more likely that a user's chosen password would not satisfy the password restrictions imposed by many sites. Extending our implementation to map each key to $3 - 4$ symbols would require more investigation into defining `keysyms` to consist of multiple characters. Older versions of Ubuntu contain a file `/usr/include/X11/keysymsdef.h` in which new such mappings could be defined. However, this file does not exist in Ubuntu 18.04.

C. Mac

In Mac OS X, custom keyboard layouts can easily be added via the `.keylayout` file type. The general format of the `.keylayout` is not particularly well documented, but one can find good examples of custom layouts files in [11]. Additionally, many Mac-specific tools exist for creating `.keylayout` files, including Ukelele [10]. After creating a `.keylayout` file, a user can add it to their machine by moving the file to the `/Library/Keyboard Layouts/` and restarting their computer.

Unfortunately, due to the addition of Secure Input in newer Mac OS X distributions – described in more depth here [12] – using a custom keyboard layout while entering a password is not allowed. Specifically, the OS will automatically switch and lock your keyboard to the standard US keyboard whenever it detects a user is inputting text into a "secure field" (i.e. a password text box). Therefore, a user would need to type their HardPass-assisted password in a plaintext field and then copy it into the password field – reducing both usability and security. For this reason, we decided to not implement a Mac-specific version of HardPass.

V. HardPass Web-server

Note: The web service described in the following section is hosted on Heroku and can currently be viewed at the following link: https://hardpass.herokuapp.com. Please note that the website may take up to 10 seconds to load because we are using the free tier of Heroku hosting.

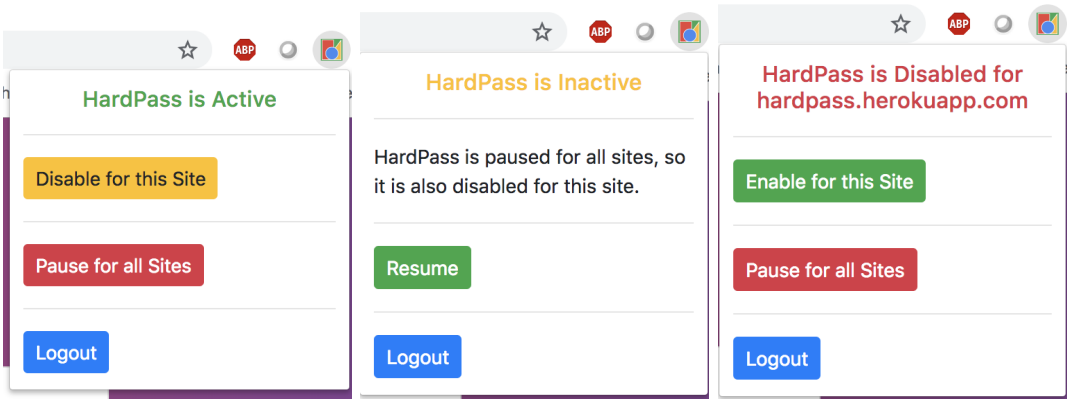
In addition to the purely local solution offered by Keyboard Drivers, we built a cloud-based service that would automatically sync across multiple devices and allow users to seamlessly use a unique keyboard mapping for each one of their accounts. As a proof-of-concept, we implemented the back-end of this service with a Python web server hosted at Heroku, and built a browser extension for Google Chrome that would communicate with our Heroku server and automatically induce the proper keyboard mapping for the proper accounts using JavaScript. Our website uses HTTPS for all communications.

Though the threat-surface of a cloud-based service is obviously larger than that of a purely local keyboard driver, there was one large security benefit that the dynamism of the cloud-based web-server offered that the static keyboard drivers could not – unique keyboard mappings for each user account. Because the website and browser extension can dynamically detect which account a user is trying to log into, and can also generate a new mapping for each unique account that a user has, the web server model can create a unique keyboard mapping for each account. This is unlike the static keyboard driver, which always maps the same character to the same keys, regardless of which account a user is logging into. This added benefit of the cloud-based service meant that even if one keyboard mapping were exposed, a user's other keyboard mappings would still be safe, thus reducing the damage that the leakage of a single mapping could inflict on a user.

A. Back-end: Server & Database

The back-end was built using the lightweight Flask web development framework, and utilizes a PostgreSQL database with two tables for storage – one to store user account information and one to store the keyboard mappings for each user and account. A detailed description of the database schema is offered below:

- Accounts
  - **ID** - Unique user ID (used for mapping users to keyboard mappings)
  - **Email** - Email address for this user
  - **Hashed Password** - SHA-256 hash of the password concatenated to the **Salt** for this user
  - **Salt** - Salt used in SHA-256 hash of this user's password
  - **Token** - Random hex string of 128 characters, used by Chrome extension to "log in" as user and fetch keyboard mappings
  - **Created** - Date and time that this account was created
- Keyboard Mappings
  - **ID** - Unique keyboard mapping ID
  - **User ID** - Unique ID of user who owns this keyboard mapping
  - **Mapping** - String encoding the keyboard mapping for this user and this web service
  - **Website** - Domain name associated with this keyboard mapping
  - **IV** - Initialization vector used for encrypting the **Mapping** field
  - **Created** - Date and time that this keyboard mapping was created
  - **Update** - Date and time that this keyboard mapping was last updated



(a) Active (b) Inactive (c) Disabled for Current Site

Fig. 3: Screenshots of the HardPass Chrome browser extension user interface

Though there were fields in the **Keyboard Mappings** table that allowed for editing keyboard mappings already created and for encrypting the mapping stored in the database, as a simple proof-of-concept prototype we did not fully implement this functionality to speed up debugging and development. Although the few simple lines of code needed for doing so have been implemented (but commented out) in *app.py*.

In addition to storing user data, the back-end was tasked with dynamically generating unique keyboard mappings for each website that a user has an account for. Each time a user loaded a new webpage with a password field, the browser extension that we built would detect this password field and then send a request to the back-end for the mapping corresponding to this domain. The back-end would then check the PostgreSQL database for an existing keyboard mapping for this user and domain. If it found one, then it would simply decode the mapping and return it to the browser extension. If one did not already exist, however, then it would loop through every possible key on a standard user's keyboard (i.e. characters with ASCII codes between 32 and 126, inclusive), and then randomly choose with replacement 3-4 characters that each key would map to. It then saved this mapping in the database before returning it to the front-end browser extension. This is implemented in the function $generateMapping$ in *app.py*.

B. Front-end: Browser Extension

Besides the front-facing website itself, the only aspect of HardPass that with which the user interacts is the browser extension that we built for Google Chrome. Once installed, the user simply enters his HardPass credentials into a form provided by the extension. This will then verify the user with the back-end and receive a token in return, which allows the extension to then act on behalf of the user. This token is then stored in the local storage that Chrome provides for extensions ($chrome.storage.local$) so that future requests from the extension will be able to access this unique per-user token.

The extension can be turned on and off dynamically by the user by simply clicking on the extension's icon (located to the right of Chrome's "Omnibox") and clicking the "Pause" button. The extension can also be disabled for specific websites by hitting the "Disable for Site" button, which will turn HardPass off for all webpages belonging to that domain. Both of these changes can be reversed with the "Resume" and "Enable" buttons, respectively.

When not disabled or paused, the HardPass extension will always be active. This means that every time the browser loads a new webpage, the HardPass extension will use JavaScript to 1) check if the extension is logged into HardPass by reading the "token" field of $chrome.storage.local$, 2) if a valid "token" is found, sending a request to the HardPass server to return the mapping for that user and the domain that the currently loading webpage belongs to, and then 3) once the webpage is fully loaded, scan its DOM for any password fields. Once the extension detects a password field, it will bind a callback function to the $keyup$ event for that field, so that every time the user presses a key in that password field the extension will be able to react. Once a $keyup$ event is triggered, the HardPass extension first checks to see whether a valid mapping for this user and domain was returned by its previous request to the HardPass server. If a valid mapping was returned, then it will take whatever character the user just entered into that field, remove it, and then insert the 3-4 character mapping for that character into the field. Thus, when the user submits the form, the value of the password field will be the mapped password for that user.

VI. Performance Analysis

In this section, we will discuss the performance of the HardPass system in three ways: 1) measure the relative password complexity of HardPass generated passwords vs. plaintext versions, 2) rate HardPass as a password scheme in categories explained in [1], and 3) discuss the security vulnerabilities of the web-server.

A. HardPass Password Complexity

Generally, password entropy or complexity is thought to be a function of password length, password domain, and password irregularity. Namely, a good password should be

SplashData Top25	Time (secs)	Ubuntu Version	Time (secs)	Windows Version	Time (secs)
123456	0	|P0Z0W	0.7	y&o&)wDY#f76⋆l8z4⋆Wz1	10^{28}
Password	0	0wZZR-l	10^5	e6&⋆9xD5##2e##2eSJ57%⋆⋆7Jc4!2546	10^{49}
12345678	0	|P0Z0Wkc	10^5	y&o&)wDY#f76⋆l8z4⋆Wz134@U&UZ(	10^{43}
qwerty	0	"RF-35	3	$⋆8SJ57m(32Jc4!fe0$QXM	10^{29}
12345	0	|P0Z0	0.013	y&o&)wDY#f76⋆l8z4	10^{19}
123456789	0	|P0Z0Wkc=	10^7	y&o&)wDY#f76⋆l8z4⋆Wz134@U&UZ(28i	10^{49}
letmein	0	(F3eF3%	10^3	08⋆Tm(32fe0$iˆYUm(32!5w2))G$	10^{40}
1234567	0	|P0Z0Wk	10^3	y&o&)wDY#f76⋆l8z4⋆Wz134@U	10^{35}
football	0	-3bw((	10^3	ˆ⋆#3%⋆⋆7%⋆⋆7fe0$Yn9@9xD508⋆T08⋆T	10^{49}
iloveyou	0	3(TF5A	100	!5w208⋆T%⋆⋆7%3(5m(32QXM%⋆⋆7⋆7X)	10^{46}
admin	0	wle3%	1	9xD52546iˆYU!5w2))G$	10^{25}
welcome	0	RF(eFR	10^3	SJ57m(3208⋆Tx$J%⋆⋆7iˆYUm(32	10^{38}
login	0	(V3%	3	08⋆T%⋆⋆7#36!5w2))G$	10^{23}
abc123	0	wbw|P0	7	9xD5Yn9@x$Jy&o&)wDY#f	10^{28}
starwars	0	Z3w-Rw-Z	10^5	##2efe0$9xD5Jc4!SJ579xD5Jc4!##2e	10^{49}
123123	0	|P0|P0	0.7	y&o&)wDY#fy&o&)wDY#f	10^{25}
dragon	0	l-wV%	10	2546Jc4!9xD5#36%⋆⋆7))G$	10^{31}
passw0rd	0	zwZZRa-l	10^3	n⋆J99xD5##2e##2eSJ574)n0Jc4!2546	10^{28}
master	0	ewZ3F-	10^2	iˆYU9xD5##2efe0$m(32Jc4!	10^{35}
hello	0	HF((	0.03	1z#wm(3208⋆T08⋆T%⋆⋆7	10^{25}
freedom	0	--FFle	100	ˆ⋆#3Jc4!m(32m(322546%⋆⋆7iˆYU	10^{40}
whatever	0	RHw3FTF-	10^4	SJ571z#w9xD5fe0$m(32%3(5m(32Jc4!	10^{49}
qazwsx	0	"wZRZ8R	10^2	$⋆89xD5G5T$SJ57##2eWoj	10^{29}
trustno1	0	3-AZ3%|	10^7	fe0$Jc4!⋆7X)##2efe0$))G$%⋆⋆7y&o	10^{46}

TABLE I: SplashData Top25 vs. HardPass Password Cracking Time

long, use "special characters," and not contain recognizable English words or phrases. Unfortunately, most don't adhere to these general guidelines. In fact, the top 25 most used passwords account for roughly 10% of all passwords [7]. In Table I, we compare the complexity of the character-to-character and character-to-string HardPass systems (as detailed in the Randomized Keyboard section). We define complexity in this context as simply the approximate time to crack the password using a state-of-the-art password cracker. Fortunately, the password manager Dashlane provides an open-source tool that can do this calculation for us [8]. It is clear from the table that the HardPass system (especially the character-to-string mapping) way outperforms the simple plaintext passwords – on average takes on the order of a septillion years to crack one of the Top25 passwords converted using the HardPass randomization.

B. HardPass Benefit Evaluation

In order to evaluate the benefits of the HardPass system objectively as possible, we turn to the evaluation method proposed in [1]. The paper splits the benefits into three categories: Usability, Deployability, and Security. Table I shows the benefits of the HardPass system compared with the benefits of the simple, text-based password system as well as LastPass. We briefly explain HardPass' rating in each category:

1) Memorywise-Effortless: Users do have to remember their passwords.

2) Scalable-for-Users: The more websites a user has accounts for, the more passwords they may need to remember.

3) Nothing-to-Carry: User doesn't need to carry an additional device.

4) Physically-Effortless: Does requite physically entering the password.

5) Easy-to-Learn: System is pretty much the same as entering in a normal password.

6) Efficient-to-Use: Pretty much as quick as typing in a normal password.

7) Infrequent-Errors: As many errors as normal text-based passwords.

8) Easy-Recovery-from-Loss: As easy as normal text-based passwords.

9) Accessible: Same as text-based passwords.

10) Negligible-Cost-per-User: Same as text-based passwords.

11)Server-Compatible: Same as text-based passwords.

12) Browser-Compatible: Web-based version only for Chrome at the moment.

13)Mature: Not as mature as text-based, but the passwords are sent to the server as text-based passwords, so quasi.

14) Non-Proprietary: Not charging anything and open-source.

15)Resilient-to-Physical-Observation: Same as text-based passwords.

16) Resilient-to-Targeted-Impersonation: Since per-user mapping, this is not really possible.

17)Resilient-to-Throttled-Guessing: With the character-to-string variant, this appears true.

18)Resilient-to-Unthrottled-Guessing: This is the same as LastPass – both employ high entropy passwords.

19) Resilient-to-Internal-Observation: Same as text-based passwords.

20)Resilient-to-Leaks-from-Other-Verifiers: With the web-based client, each website has a unique mapping.

Scheme	*Memorywise-Effortless*	*Scalable-for-Users*	*Nothing-to-Carry*	*Physically-Effortless*	*Easy-to-Learn*	*Efficient-to-Use*	*Infrequent-Errors*	*Easy-Recovery-from-Loss*	*Accessible*	*Negligible-Cost-per-User*	*Server-Compatible*	*Browser-Compatible*	*Mature*	*Non-Proprietary*	*Resilient-to-Physical-Observation*	*Resilient-to-Targeted-Impersonation*	*Resilient-to-Throttled-Guessing*	*Resilient-to-Unthrottled-Guessing*	*Resilient-to-Internal-Observation*	*Resilient-to-Leaks-from-Other-Verifiers*	*Resilient-to-Phishing*	*Resilient-to-Theft*	*No-Trusted-Third-Party*	*Requiring-Explicit-Consent*	*Unlinkable*
Web passwords			●		●	●	○	●	●	●	●	●	●	●		○						●	●	●	●
LastPass	○	●	○	○	●	●	●	○	●	○	●		●		○	○	○	○		○	●	●		●	●
HardPass			●		●	●	○	●	●	●	●	○	○	○		●	●	○		○	○	●	○	●	●

● = offers the benefit; ○ = almost offers the benefit; *no circle* = does not offer the benefit.

TABLE II: Web Passwords vs. LastPass vs. HardPass

21) Resilient-to-Phishing: Quasi, given the web-based scheme, as keyboards are unique for each website.

22)Resilient-to-Theft: Same as text-base passwords.

23)No-Trusted-Third-Party: Quasi, as the local version is impervious.

24) Requiring-Explicit-Consent: Same as text-based passwords.

25)Unlinkable: Same as text-based passwords.

C. Cloud-based HardPass: Design Challenges/Trade-offs

There are several important challenges and trade-offs that we encountered when designing a cloud-based version of HardPass. First, there is currently a race condition between the asynchronous request sent by the browser extension for the keyboard mapping (step (2) of the description in the previous section) and the action itself of the user inputting his password into a password field on the loaded webpage. Though not implemented in our prototype, this could be resolved by having HardPass download all existing keyboard mappings for a user after the user first logs into the extension, and only notifying the user that HardPass is "Active" once these mappings have been successfully downloaded. That would avoid the need to constantly check for existing mappings upon every page load. When a user then wanted to use HardPass on a website not previously seen, HardPass would need to inject a popover on top of the password field for that new website with a loading icon of sorts, informing the user that HardPass was busy querying the server for a new mapping and that the extension wasn't ready yet. The HardPass extension could also set $disable = "True"$ on the password field itself to prevent user input until it received the correct mapping from the server. Implementing these steps would solve the race condition but would require that the HardPass server respond promptly to requests, otherwise risk angering users who must wait long times for their mappings to be returned.

Another challenge encountered when building the extension was defining what exactly the mapping between URLs and user accounts entailed. Essentially, mapping the syntax of a URL to the semantics of having an account with a username and login for a specific service was much more nuanced than originally thought. For example, the domains `google.com`, `google.co.uk`, `drive.google.com`, and `mail.google.com` should all have the same keyboard mapping for a specific user, for they are all linked to the same central Google account for that user. Because there is now a tremendous variety of top-level domains, and there is no clear-cut parsing pattern that exists to separate `google` from the aforementioned domains, a script that wanted to map all `google.com` domains and subdomains to `Google` would have to leverage the Public Suffix List provided freely by Mozilla. There are JavaScript libraries that implement this, including $tldjs$, which we would like to incorporate into our extension in the future. However, for our current implementation of our extension, we instead use a slightly more conservative URL decoding scheme to remove subdomains that is based on the highest level domain for which the extension can set a cookie.

Another challenge was persisting a user's login state after the user logs into HardPass through the browser extension. We settled on generating a unique, random 128-character long hex string "token" that would be associated with each user account and allow any request that had the correct "token" to access the keyboard mappings for that user. This avoided the need to store the user's plaintext password in local storage, which would limit the damage of any memory-based attacks on the user's Chrome browser by not revealing the user's actual password (and therefore only allowing an adversary to access that user's keyboard mappings). However, a negative "side-effect" of this design choice was that it was then impossible to encrypt keyboard mappings in the PostgreSQL database with user passwords, for the HardPass extension was purposely designed to not store the user's plaintext HardPass password in local memory and to instead rely on the random "token" to act on behalf of the user. This meant that if the keyboard mapping returned from the server were encrypted with that user's HardPass password, then there would be no way for the HardPass browser extension to decrypt it without continuously prompting the user to re-enter his HardPass password (a usability nightmare).

Now, one might suggest that a simple solution would be to just encrypt the keyboard mappings of the database with the user's unique "token" value, which is a value that the browser extension would have access to. The issue with this, however, is that the "token" value is also stored in the database,

which means that an adversary that hacked into the database and read its contents would be able to instantly decrypt the keyboard mappings for each user, for the key used to encrypt each mapping would simply be the value stored in the "token" field of the accounts table. Separating the two tables into separate databases on separate physical servers, like LastPass does with its Authentication Keys and encrypted Vaults, would help mitigate this problem but would not completely solve it.

Thus, there is an inherent security trade-off between A) storing a user's plaintext HardPass password locally and then being able to encrypt his keyboard mappings on the server in such a way that an adversary who read the contents of the database would not be able to decrypt (for HardPass does not store user passwords on its server), or B) using an easily revocable, random "token" for user log ins but at the expense of a more completely secure keyboard mapping encryption scheme and more complicated isolation of database tables needed on the server-side. For our prototype, we settled on the latter approach, but were not able to set up two physically distinct servers since we lacked such fine-grained control over our storage systems when using Heroku's free tier.

Another key challenge faced when designing the JavaScript code to inject mapped characters into a password field was the issue of deleting input, whether though a backspace or holding `CMD+A` before typing. Because the user has no idea how many characters were mapped to each of his keypresses, let alone which characters were mapped, it didn't make sense for the HardPass extension to allow the user to delete one character at a time if he had mistyped a character. For example, assume `h` maps to `HHH`, `j` maps to `JJJJ`, and `i` maps to `III`. If the user typed `hj`, which was mapped to `HHHJJJJ` by HardPass, then hitting the backspace button and then typing `i` to make the password field correspond to the mapping for `hi` would not result in the desired output. Removing one character would lead to the password field containing the value of `HHHJJJ`, and then typing an `i` would result in the password field reading `HHHJJJIII` instead of the desired `HHHIII`. Our solution was to simply have the backspace keypress clear out the entire contents of the input field and force the user to start typing their password over, which is not that severe of a usability limitation since most times a user mistypes his password he is not sure where he mistyped it anyway, and thus will start over. Additionally, because "strong" passwords with HardPass can actually be relatively simple to type (i.e. a short, easily-typed string like `password` can map to `T@/5AD*Wn!y{R,vF*1nvmI9$lL")`, users may only need to type in a few characters anyways when logging in.

In terms of security, the last main challenge that would need to be further explored before deploying HardPass in a production environment would be to securitize the server itself using standard web security procedures. Though we already use HTTPS for all web communications, hash passwords and salt them with SHA-256, use the *SQLAlchemy* Python library for communicating with our database to automatically escape special characters and thereby avoid SQL injection, etc., there are many features that, while not necessarily unique to a cloud-based password manager and thus a bit tangential to the main point of this paper (which was developing a better paradigm for thinking about what information password managers should store), would be essential for any public-facing web service. For example, the 'token" that is used to authenticate the Chrome extension after a user has logged in should expire after a certain number of days to prevent stale tokens from being able to log in on behalf of a user. Additionally, the keyboard mappings themselves that are stored in the database should be encrypted in some way, although for reasons outlined above we made the choice not to do this for the purposes of our proof-of-concept. And since our website has login functionality, we should also implement some form of throttling to prevent brute-force attacks. This could be easily achieved by adding a table to our PostgreSQL database that kept track of the timestamp and user ID for every login attempt, and if there were 5 or so login attempts in the past 5 minutes, refuse to check whether the current login attempt was valid. In terms of usability, adding a "Remember me" button to our login form, as well as "Forgot password" functionality would be helpful for users, although we would also need to be aware of the security implications of adding such functionality.

VII. Conclusion

In this paper, we have described HardPass, a keyboard-centric password manager. We have designed the system to be as usable as standard, text-based password schemes, while adding better security guarantees. We achieve this by creating for each user a randomized keyboard that, when used, maps keyboard input into randomized output. Using our key-to-string keyboard randomization specifically, we were able to show that our system drastically improves protection against password cracking attacks – making even the top 25 most popular passwords virtually uncrackable. We also provide detailed analysis of HardPass along the axes of usability, deployability, and security. As preliminary implementations of our core ideas, we present versions of HardPass for Ubuntu and Windows machines, as well as a working proof-of-concept HardPass web-server that integrates the randomized keyboard scheme into a Google Chrome plugin.

VIII. Future Work

Despite having a working implementation of the HardPass system, it is still only a proof-of-concept. As we've mentioned throughout the paper, there are numerous areas that would benefit from further exploration. Some of these areas include:

1) Implement HardPass system across additional platforms, including iOS, Android, and others.
2) Investigate syncing mechanisms/protocols between the web-based system and the system-specific systems.
3) Explore other, non-xkb methods in order to implement the character-to-string keyboard randomization on Linux.
4) Further investigate methods to ensure HardPass-mapped passwords deterministically meet password security guidelines (i.e. HardPass passwords necessarily include an upper-case letter, number, and special character) without compromising on entropy.
5) Explore potentially more secure methods of randomization, including successive character hashing (i.e. mapping characters to hashes based on the previously hashed character) so that two of the same characters won't necessarily map to the same output. This

will drastically help in preventing frequency-based attacks.

6) Explore ways to make the HardPass web-server more secure – as described in the *Cloud-based HardPass: Design Challenges/Trade-offs* section.

ACKNOWLEDGEMENTS

We would like to especially thank Dr. James Mickens for his help, general wisdom, and timeless insights as we came up with and actually completed this final project – as well as for being such an outstanding professor all semester! We also would like to acknowledge and thank the entire CS263 course staff for all of their hard work in creating a smooth and extremely successful and enjoyable class.

REFERENCES

[1] J. Bonneau, C. Herley, P. C. van Oorschot, and F. Stajano, The quest to replace passwords: A framework for comparative evaluation of web authentication schemes, University of Cambridge Computer Laboratory, Tech Report 817, 2012, www.cl.cam.ac.uk/techreports/UCAM-CL-TR-817.html.

[2] R. Morris and K. Thompson, Password security: a case history, Commun. ACM, vol. 22, no. 11, pp. 594597, 1979.

[3] A. Adams and M. Sasse, Users Are Not The Enemy, Commun. ACM, vol. 42, no. 12, pp. 4146, 1999.

[4] C. Herley and P. C. van Oorschot, A research agenda acknowledging the persistence of passwords, IEEE Security & Privacy, vol. 10, no. 1, pp. 2836, 2012.

[5] Saranga Komanduri, Richard Shay, Patrick Gage Kelley, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Serge Egelman. 2011. Of Passwords and People: Measuring the Effect of Password-Composition Policies. In Proc. CHI11: 29th Annual ACM Conference on Human Factors in Computing Systems. http://doi.acm.org/10.1145/1978942.1979321

[6] J. Jose, T. T. Tomy, V. Karunakaran, Anjali Krishna V, A. Varkey and Nisha C.A., "Securing passwords from dictionary attack with character-tree," 2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET), Chennai, 2016, pp. 2301-2307. doi: 10.1109/WiSPNET.2016.7566553

[7] Kristen Korosec, "The 25 Most Common Passwords of 2017 Include Star Wars," Fortune, 2017. http://fortune.com/2017/12/19/the-25-most-used-hackable-passwords-2017-star-wars-freedom/

[8] Dashlane Password Tool, https://howsecureismypassword.net/

[9] "Password special characters," OWASP. https://www.owasp.org/index.php/Password_special_characters

[10] Mac OS X Keyboard Layout Editor. https://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&id=ukelele

[11] Custom Mac OS X Keyboard Layouts. https://github.com/mathiasbynens/custom.keylayout

[12] Mac Secure Input. https://smilesoftware.com/textexpander/secureinput

[13] "Microsoft Keyboard Layout Creator 1.4." https://www.microsoft.com/en-us/download/details.aspx?id=22339

[14] Igor Levicki, "HOWTO: Build keyboard layouts for Windows x64," 2006. https://levicki.net/articles/2006/09/29/HOWTO_Build_keyboard_layouts_for_Windows_x64.php

[15] "XKB." https://www.x.org/wiki/XKB/

[16] "LastPass: Home Page." https://www.lastpass.com/

[17] "LastPass: How it Works." https://www.lastpass.com/how-lastpass-works

[18] Ravenscraft, Eric. "LastPass Hacked, Change Your Master Password Now," 2015. LifeHacker. https://lifehacker.com/lastpass-hacked-time-to-change-your-master-password-1711463571

[19] Mickens, James. Lecture on Passwords. 10/16/18.

[20] S. Chiasson, E. Stobert, A. Forget, R. Biddle, and P. C. van Oorschot, "Persuasive cued click-points: Design, implemen- tation, and evaluation of a knowledge-based authentication mechanism, IEEE Trans. on Dependable and Secure Com- puting, vol. 9, no. 2, pp. 222235, 2012.

[21] R. Biddle, S. Chiasson, and P. C. van Oorschot, "Graphical Passwords: Learning from the First Twelve Years, ACM Computing Surveys, vol. 44, no. 4, 2012.

[22] A. K. Jain, A. Ross, and S. Pankanti, "Biometrics: a tool for information security, IEEE Transactions on Information Forensics and Security, vol. 1, no. 2, pp. 125143, 2006.

[23] A. Ross, J. Shah, and A. K. Jain, "From Template to Image: Reconstructing Fingerprints from Minutiae Points, IEEE Trans. Pattern Anal. Mach. Intell., vol. 29, no. 4, pp. 544560, 2007.

[24] T. Matsumoto, H. Matsumoto, K. Yamada, and S. Hoshino, "Impact of artificial gummy fingers on fingerprint systems, in SPIE Conf. Series, vol. 4677, Apr. 2002, pp. 275289.

[25] Salil Pravin Vadhan, "A Study of Statistical Zero-Knowledge Proofs," 1999. https://people.seas.harvard.edu/~salil/research/phdthesis.pdf

[26] R. Jhawar, P. Inglesant, N. Courtois, and M. A. Sasse, "Make mine a quadruple: Strengthening the security of graphical one-time pin authentication, in Proc. NSS 2011, pp. 8188.

[27] Ometov, Aleksandr et al. "Multi-Factor Authentication: A Survey. Cryptography 2 (2018): 1. https://www.mdpi.com/2410-387X/2/1/1/pdf

[28] https://www.yubico.com/

[29] Preston Ackerman. 2014.Impediments to adoption oftwo-factor authentication by home end-users. TechnicalReport. SANS Institute.