Security and Privacy Failures in Popular 2FA Apps

Conor Gilsenan
UC Berkeley / ICSI

Fuzail Shakir
UC Berkeley

Noura Alomar
UC Berkeley

Serge Egelman
UC Berkeley / ICSI

Abstract

The Time-based One-Time Password (TOTP) algorithm is a 2FA method that is widely deployed because of its relatively low implementation costs and purported security benefits over SMS 2FA. However, users of TOTP 2FA apps face a critical usability challenge: maintain access to the secrets stored within the TOTP app, or risk getting locked out of their accounts. To help users avoid this fate, popular TOTP apps implement a wide range of backup mechanisms, each with varying security and privacy implications. In this paper, we define an assessment methodology for conducting systematic security and privacy analyses of the backup and recovery functionality of TOTP apps. We identified all general purpose Android TOTP apps in the Google Play Store with at least 100k installs that implemented a backup mechanism ($n = 22$). Our findings show that most backup strategies end up placing trust in the same technologies that TOTP 2FA is meant to supersede: passwords, SMS, and email. Many backup implementations shared personal user information with third parties, had serious cryptographic flaws, and/or allowed the app developers to access the TOTP secrets in plaintext. We present our findings and recommend ways to improve the security and privacy of TOTP 2FA app backup mechanisms.

1 Introduction

Though passwords are still the predominant authentication method [17, 40], it is well established that most people cannot create strong passwords [32, 33, 68]. Two-factor authentication (2FA)—logging in with a combination of at least two of something you know, something you physically have, and something you are—has consistently been shown to significantly increase the security of online accounts compared to the use of a password alone [24, 71]. The Time-based One-Time Passwords (TOTP) algorithm [8] is a 2FA method that is widely deployed, including at some of the largest sites on the Internet [1]. In addition to a username/password, TOTP requires the user to enter a one-time password (OTP) to login. In practice, these OTPs are typically 6-digit codes that are generated by a *TOTP authenticator app*[1] installed on the user's phone/mobile device (Figure 1), which change to a unique value periodically (typically, every 30 seconds).

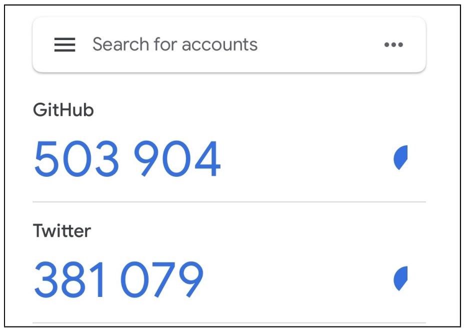


Figure 1: Screenshot of the Google Authenticator TOTP app.

TOTP 2FA is often promoted as a more secure method of 2FA than SMS 2FA [25], which inherits many vulnerabilities from the telecommunication networks on which SMS operates [44, 45]. In contrast, TOTP apps can generate OTPs without any network connection by using a locally-stored secret obtained from the server during setup. However, TOTP 2FA places a critical usability burden on users: maintain access to these shared secrets, or risk getting locked out of their account(s). In practice, these secrets are routinely lost; people lose their devices, buy new ones, or uninstall their TOTP apps. Without the TOTP shared secret, the user cannot generate the OTPs required to authenticate and could face account lockout.

To combat this usability nightmare, many TOTP apps provide custom backup mechanisms to help users recover from device loss. These backup features are critical usability enhancements, but it is understudied how they impact the security and privacy of the millions of people who use these TOTP apps on a daily basis. Thus, our goal was to comprehensively investigate the security and privacy issues that exist in the backup mechanisms of the most used TOTP apps. Our research questions included:

[1]For brevity, we refer to "TOTP apps" throughout this paper.

RQ1 What personal information, if any, is leaked to the company that develops the TOTP app, or other third parties, as a result of using the TOTP backup mechanisms?

RQ2 What is the risk of an attacker obtaining a TOTP backup?

RQ3 What is the risk of an attacker compromising the TOTP secret(s) stored within an obtained TOTP backup?

To answer these questions, we identified all general purpose Android TOTP apps in the Google Play Store with at least 100k installs that implemented a backup mechanism ($n = 22$). For each app, we manually registered TOTP accounts and exercised the available backup mechanism(s) while recording plaintext network traffic. If we determined that encryption beyond TLS was used, we performed a cryptanalysis of the app to determine how the implementation and/or use of cryptography impacted the security of the TOTP backup.

While TOTP 2FA is often billed as a security enhancement relative to passwords and SMS-based 2FA [25], our analysis found that most backup strategies end up placing trust in the same technologies that TOTP 2FA is meant to supersede: passwords, SMS, and email. The most commonly-supported backup mechanisms were syncing encrypted TOTP backups automatically to the cloud and leveraging the built-in Android Auto Backup system. Alarmingly, two apps sent the plaintext TOTP secrets to the app developer. Apps that encrypted TOTP backups had a wide range of serious vulnerabilities. Several apps ($n = 4$) sent both the encrypted backup and the encryption key (or the password from which it was derived) to the app developer, allowing them to decrypt the backup and read its content. Nearly all apps that encrypted TOTP backups derived a key from a user-provided password, but most also had severely inadequate password policies and/or used weak methods of deriving a key from the password. As a result, the ciphertexts were vulnerable to trivial offline attacks. More than half ($n = 12$) of the apps we analyzed allowed the user to manually or automatically create plaintext TOTP backups, but only two of those apps warned the user about the risk of doing so. Our contributions are as follows:

- We present a methodology for evaluating the security and privacy properties of backup mechanisms in TOTP apps, using both dynamic analysis and cryptanalysis.
- We show that many popular 2FA apps use vulnerable security mechanisms, thereby exposing TOTP secrets.
- We show that many popular TOTP apps leak user information, including the names of the websites/services they use and their account usernames on those platforms.

2 TOTP Overview

While enabling TOTP 2FA, the website/service typically instructs the user to install a TOTP authenticator app (often recommending one or more specific apps) and scan a QR code displayed in their browser. Doing so shares three key pieces of information with the TOTP app:

1. *Issuer*: the name of the website/service on which the user is currently setting up TOTP 2FA;
2. *Label*: the user's account username; and
3. *Secret*: a random secret generated by the website/service that is unique to the user's account.

The *secret* is used by the app to generate and display the periodically-changing OTPs, while the *issuer* and *label* are used to visually indicate which OTPs go with which accounts.

The TOTP RFC [8] specifies how the client (i.e., the app) and server (i.e., the website/service) should utilize a shared secret to generate and validate a deterministic OTP during authentication. However, many practical implementation considerations are out of its scope. For example, it does not suggest any mechanism to securely transfer the shared secret from server to client. A publication by Google [5] has become the *de facto* standard to fill this gap and defines how to transfer the issuer, label, secret, and other data using QR codes.

3 Related Work

In this section, we provide an overview of related work on account recovery and mobile app analysis.

3.1 2FA and Account Recovery

It is well established that a major road block to 2FA adoption in general is the fear of losing physical authenticators. FIDO2 [31], which is supported by all major browsers, provides an authentication mechanism that relies on physical possession of a device and is resistant to phishing attacks, but research has consistently found that users are concerned about losing their authenticator and getting locked out of their accounts [19,21,29,47,53,64]. After evaluating 12,500 crowdsourced comments about 5 prevalent MFA mobile apps, Das et al. [23] called for improvements to backup and recovery mechanisms to eliminate account lockout concerns. As a result, websites commonly implement alternative methods of authentication that do not require physical possession of a device: security questions, email, and SMS.

Email Due to known security and usability issues with "knowledge-based recovery questions" [16, 24, 57, 67], websites often rely on email to complete account recovery. Li et al. [46] found that allowing password recovery via email alone was nearly ubiquitous among the Alexa Top 500. Doerfler et al. [24] analyzed large, real-world data sets from Google and found that sending an OTP to a secondary email address encountered significant usability and security issues. Raponi and Di Pietro [58] proposed a scheme to address the risk of internal attackers at email providers hijacking email accounts.

SMS Doerfler et al. found that SMS 2FA was highly effective against automated attacks by bots, but only prevented 76% of targeted attacks [24]. While SMS 2FA has one of the most usable recovery processes of any device-based 2FA (because SIM cards can be replaced by service providers), this property reveals that SMS 2FA does not actually rely on physical possession. SMS 2FA is not considered a secure authentication mechanism due to the ease of hijacking phone numbers [20, 44]. After observing the prevalence of recycled phone numbers actively associated with the accounts of the previous owner, Lee and Narayanan [45] concluded that websites/services "...should no longer equate a correctly-entered SMS passcode with successful user authentication."

TOTP Gilsenan et al. previously identified several security and privacy issues in the Authy Android app [35], but research is needed to determine whether those issues are pervasive across other popular TOTP 2FA apps.

Polleit and Spreitzenbarth [56] reviewed 16 Android TOTP apps based on user ratings. They reported where and how each app stored the TOTP secrets on the device and performed a basic analysis of network traffic captured with mitmproxy. In contrast, our analysis encompassed the entire backup and recovery workflow and we performed a significantly more thorough review of the traffic generated by apps in our dataset. Leveraging user ratings as a filtering heuristic caused them to review several apps with small install counts, some less than 1,000. In our work, we analyzed *all* TOTP 2FA apps in the Google Play Store with over 100,000 installs that supported a backup mechanism; our dataset included 22 apps that comprised over 180 million installs.

Ozkan and Bicakci [54] reviewed 11 popular Android TOTP apps and showed that they could read the plaintext TOTP secrets from storage for 5 apps and from memory for 7 apps. However, they considered a different threat model than us; the attacker possessed the device and had root access.

3.2 Mobile App Analysis

Over the past decade, many researchers have examined the privacy and security implications of mobile apps (e.g., [30, 42, 70, 72]). Most current approaches to analyzing mobile app behaviors rely on static analysis [34, 37, 43, 80]: reverse-engineering sequences of program code to infer application behavior. This method invariably falls short in that it can only detect what behaviors or capabilities a program *might* have, and not *whether* and to *what extent* a program actually engages in these behaviors. For example, it is impossible in general to predict the full set of branches that a program will take. Other techniques like taint tracking [28], which modify application data so that it can be observed traversing through an application, have other problems (e.g., impacting application stability) [18].

A more recent approach is adding instrumentation to the Android operating system itself to monitor third-party apps' access to sensitive user data at runtime [73, 77–79]. This allows researchers to examine a wide range of app behavior, including app-associated network traffic. Prior solutions to monitoring mobile app transmissions generally involve using proxy software (e.g., Charles Proxy,[2] Wireshark,[3] mitmproxy[4]) and suffer from serious shortcomings. First, *all* device traffic is usually routed through the proxy, without automatically identifying which traffic came from which app running on the device. While some traffic may contain clues (e.g., content and headers that may identify apps, e.g., HTTP `User-Agent` headers), other traffic does not, and disambiguating the identity of the app is a laborious and uncertain process [59]. Second, proxies cannot automatically decode various obfuscations, including TLS with certificate pinning. Instead, by capturing traffic from the monitored device's OS, these problems can be avoided: certificate pinning can be bypassed, decryption keys can be extracted from memory, and individual sockets can be mapped to process names (providing strong attribution to individual apps). We utilized this type of platform-level instrumentation in our analysis.

4 Methods

In this section, we explain how we selected the 22 TOTP 2FA Android apps to analyze and the systematic procedure that we used to analyze each app.

Throughout the rest of the paper, we use the term *plaintext* to refer to data that is not end-to-end (e2e) encrypted before leaving the app. We use the term *encrypted* to indicate that the app applied e2e encryption prior to transmission (i.e., regardless of whether TLS was used for transmission; all apps tested used TLS when transmitting backup data).

While there are myriad ways an attacker can compromise a device that is in their physical possession, we consider these attacks out of scope. Instead, our analysis focused on the threat model of an attacker obtaining the data contained in a TOTP backup (e.g., the secret, label, and issuer) once it leaves the local device. We assume that the Android device is free of malware and has the latest security updates applied.

4.1 App Selection

In November and December 2021, we identified as many TOTP apps in the Google Play Store as possible. To start, we created a list of core search terms that we knew from personal experience would return TOTP apps (e.g., "totp", "2fa", "two factor", "mfa", and "multifactor"). We entered each core search term into the search box on the Google Play Store and incorporated the top 5 auto-completion suggestions.[5]

We queried the Google Play Store for each unique search term and downloaded the metadata for the top 30 apps displayed in each query result set. Removing duplicates yielded

[2]https://www.charlesproxy.com/
[3]https://www.wireshark.org/
[4]https://mitmproxy.org/
[5]See Section 7 for supplemental material available online.

546 apps. We further narrowed the candidate pool by excluding 193 apps that had fewer than 100,000 installs and 263 apps whose description did not contain any of the final search terms. The remaining 90 apps were each reviewed manually to determine whether they were, in-fact, TOTP 2FA apps.

We focused exclusively on apps whose primary focus was 2FA and could be used on any site that supports TOTP. We excluded any apps that only worked on a specific service (e.g., Blizzard Authenticator). For purposes of practicality and scope, we excluded multipurpose apps that offer TOTP support amongst other functionality (e.g., password managers). Analyzing the 22 TOTP apps that satisfied our criteria took a significant amount of time and effort. While many TOTP apps include some cryptographic features that are outside of our scope (e.g., push 2FA, encrypting local storage), many password managers use cryptography much more extensively than TOTP apps, which would make analysis even more complex.

4.2 App Analysis

For each app in our data set, we downloaded the Android Package (APK) file from the Google Play Store using a non-rooted Pixel 3a phone, the Android Debug Bridge (ADB), and a custom shell script.[6] We analyzed each APK using the three phases described in the following subsections.

4.2.1 Exploring the App

The first step of analysis was to explore the app and document its various features and settings. We recorded whether the app required any personal information to use the app at all. For example, the *Twilio Authy* app required the user to provide an email address and prove control of a phone number.

We also answered a range of other questions that could be determined by using the app. What personal information, if any, is required to enable/utilize the backup mechanism? What backup mechanisms does the app support (e.g., remote backups, manual exports, transfer via QR code, etc.)? If remote backups are available, where are they stored (e.g., the developer's cloud service, Google Drive, etc.)?

If the backup mechanism required creating a password, then we attempted to manually determine the password policy, including minimum required length and use of a block list.

Finally, we created a customized checklist that enumerated exactly which actions to take within the app and which data to enter (e.g., phone numbers, email addresses, and passwords) while recording the network traffic. Each checklist included a core set of steps designed to replicate the real world actions that a user would take when first using the app, enabling its backup mechanism, and eventually executing the recovery process. We believe that a reasonable user executing these steps would not expect any of their personal information to be sent remotely in plaintext, unless they were informed.

[6]https://github.com/blues-lab/getapk

We specifically documented details about each app's recovery workflow so that we could determine which attack vectors were available to an attacker attempting to impersonate a user and obtain the TOTP backup remotely.

4.2.2 Capturing & Reviewing Network Traffic

After becoming familiar with the app, we followed the customized checklist to exercise the app's backup and recovery mechanisms in a controlled environment and noted which information, if any, was sent remotely. The goals of this phase were to (1) identify any plaintext TOTP fields in backups that were transmitted remotely and (2) record any fields in the TOTP backups that appeared to be encrypted.

We installed each app on a Pixel 3a phone running a custom version of Android 9 (initially developed in prior research [60, 63,77,78] and commercially maintained by AppCensus[7]) that recorded plaintext network traffic to file[8]. A variety of open source tools for collecting network traffic (e.g., mitmproxy,[9] Magisk,[10] and Frida[11]) can be used to verify our results and, we believe, reproduce our findings from scratch.

We ensured that each phone had a SIM/phone number and a registered Google account. For reference and completeness, the phone's screen was recorded as an mp4 video using scrcpy[12] during the network logging session. The ability to retroactively review the specific actions that led to a specific transmission proved to be an invaluable time-saving resource.

As mentioned previously, the most common method of setting up TOTP 2FA is for the user to scan a QR code using the TOTP app. We followed the de facto standard [5] to create two QR codes that encoded specific values for the issuer, label, and secret fields. Using these QR codes throughout our analysis allowed us to check whether these known values appeared in the network traffic generated by each app.

After recording, we reviewed the network traffic to identify which request/response calls were responsible for sending the TOTP backup to the remote storage service, if any. We checked whether the network traffic contained any known values, such as passwords entered to enable the backup mechanism and any of the fields encoded in our custom QR codes. Finally, we documented the value of any field that appeared to be encrypted so that it could be further analyzed in detail.

4.2.3 Performing Cryptanalysis

If the collected network traffic contained any encrypted fields, we performed a cryptanalysis of the app to determine which cryptographic primitives were used to create the ciphertext in

[7]https://appcensus.io
[8]See Section 7 for supplemental material available online.
[9]https://mitmproxy.org/
[10]https://github.com/topjohnwu/Magisk
[11]https://frida.re/docs/android/
[12]https://github.com/Genymobile/scrcpy

the TOTP backup and how they were configured. Understanding these details is critical to assessing the overall security of the app's backup mechanism because any attacker with access to the ciphertext would attempt to learn the same information to launch an offline attack.

For apps that were not open sourced, we decompiled the APK using the jadx decompiler.[13] Analyzing obfuscated code was often complex. We identified potentially-relevant code by searching for static strings used in crypto libraries (e.g., AES, PBKDF2), compared function signatures to open source crypto libraries (e.g., Bouncy Castle[14] and libsodium[15]), and manually refactored the code to improve readability.

Our cryptanalysis consisted of two main phases. In the first phase, we reviewed the code to determine how the app obtained the encryption key used to generate the ciphertext in the TOTP backup. The most prevalent backup architecture among the apps we analyzed involved deriving a symmetric encryption key from a user-provided backup password. Therefore, we documented the key derivation function (KDF) that was used and how it was configured, including whether or not a salt was used and, if so, whether it was random or static.

In the second phase, we focused on determining the cryptographic primitives used to produce the ciphertext, including which encryption ciphers were used, their modes of operation, and the method of authentication, if any. Most apps did not provide any integrity over the ciphertext, so we recorded any custom heuristics used to determine whether or not the correct backup password was provided during recovery. For example, some apps checked whether the decrypted backup was a valid JSON object. These are the same heuristics that an attacker would use in an offline attack.

Many of the apps that we analyzed implemented multiple features that relied on various methods of cryptography. Therefore, we could not assume that our observations and assumptions about the cryptography used to generate the ciphertext in the TOTP backup were correct. To prove that we had identified the correct cryptographic primitives and configurations, we implemented the app's decryption process in a separate script[16] that accepted the following inputs: (1) the ciphertext and IV from the network traffic; if applicable, (2) the password that we chose when enabling the backup mechanism, and (3) any salt passed to a KDF. We manually verified whether the decryption process was correctly implemented by looking for known values in the resulting plaintext, such as the TOTP secret from our custom QR codes.

5 Results

In this section, we explain the various backup mechanisms available in the apps that we analyzed and discuss the security and privacy implications of each.

5.1 Backup without the Network

Like the *de facto* standard [5] for initially transferring TOTP data from the website/service to the TOTP app, 7 apps (see Table 1) leveraged QR codes to allow users to optically transfer TOTP secrets between devices without the risk of sending data over the network (RQ2). Of these, several bury this feature below multiple menu layers, decreasing the odds that users will find the feature unless specifically looking for it.

Manual export via QR code is the only backup mechanism supported by Google Authenticator, the app with the most installs ($n = 100M+$) by a factor of 2 or more. Google stated that balancing security and usability was a specific motivation when it rolled out support for this feature and highlighted that "...physical access to [the] phone and the ability to unlock it" is required [39]. Google Authenticator also records an audit log that allows users to detect suspicious secret transfers.

While backups via QR code prioritize confidentiality and integrity, the mechanism falls short on availability. Scanning a QR code with another personal device is inherently a manual activity. Users must own one or more secondary devices, know their location, and physically transfer TOTP data from their primary device to their secondary device(s) each time they add a new account. We believe it is likely that many users will not perform this backup ritual reliably because prior work has found that people are often unsuccessful at regularly creating manual backups for their devices in general [61]. While periodic nudges or other reminder techniques may increase the manual backup rate, many users could face account lockout when they inevitably forget to backup some TOTP secrets and actually need to recover.

The threat model for some users may demand the security benefits of using QR codes to transfer TOTP secrets between devices, but we suspect that most users will prefer more automated solutions. To enable any type of automated backup system, some data must inevitably be sent over the network.

5.2 Remote Backups *without* Encryption

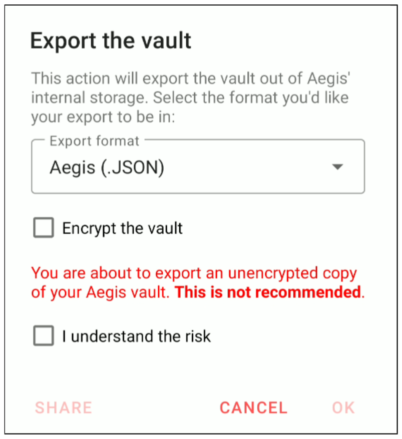
As shown in Table 1, more than half (12) of the apps in our dataset, comprising almost 8 million installs, are capable of backing up the TOTP data in plaintext. The majority of these apps (10) supported sharing plaintext backups directly via the standard Android sharing menu (e.g., via email, SMS, etc.), copying the plaintext backup to the clipboard, and/or exporting the plaintext backup to the device file system where it could be shared like any other file.

Just 3 apps supported plaintext cloud sync, which sent all of the TOTP fields (secret, label, issuer) to the cloud in plaintext. The *Latch* app automatically sent all TOTP fields to Latch servers in plaintext with no option for users to opt out. This was the only backup mechanism the *Latch* app provided to its

[13] https://github.com/skylot/jadx
[14] https://www.bouncycastle.org/
[15] https://libsodium.org
[16] See Section 7 for supplemental material available online.

Abbreviated Name	**APK id@version**	**Installs**	**Backup Mechanisms**							
			QR Codes	**Cloud Sync**		**File Export**		**Sharing**		**Android Backup**
				Plaintext	**Encrypted**	**Plaintext**	**Encrypted**	**Plaintext**	**Encrypted**	
Google Authenticator	com.google.android.apps.authenticator2@v5.10	100M+	Y	-	-	-	-	-	-	-
Microsoft Authenticator	com.azure.authenticator@v6.2204.2757	50M+	-	-	Y*	-	-	-	-	-
Duo Mobile	com.duosecurity.duomobile@v4.15.0	10M+	-	-	Y	-	-	-	-	-
Twilio Authy	com.authy.authy@v24.8.5	10M+	-	-	Y	-	-	-	-	-
Latch	com.elevenpaths.android.latch@v2.2.4	5M+	-	Y	-	-	-	-	-	-
LastPass Authenticator	com.lastpass.authenticator@v2.5.0	1M+	-	-	(Y)	-	-	-	-	-
2FAS	com.twofasapp@v3.11.0	1M+	-	Y	Y*	Y	Y	Y	Y	-
Yandex.Key	ru.yandex.key@v2.7.0	1M+	-	-	Y*	-	-	-	-	-
FreeOTP Authenticator	org.fedorahosted.freeotp@v1.5	1M+	-	-	-	-	-	-	-	Y
Authenticator	com.pixplicity.auth@v1.0.6	500k+	Y	-	-	-	-	Y	Y*	-
Salesforce Authenticator	com.salesforce.authenticator@v3.8.5	500k+	-	-	Y*	-	-	-	-	-
Code Generator	net.codemonkey.otpgeneratorapp@v6.1	500k+	-	-	-	Y	-	-	-	Y
TOTP Authenticator	com.authenticator.authservice2@v1.89	100k+	Y	-	Y*	-	Y*	Y	Y	Y
Aegis Authenticator	com.beemdevelopment.aegis@v2.0.3	100k+	-	-	-	Y	Y	Y	Y	Y
Auth0 Guardian	com.auth0.guardian@v1.5.3	100k+	-	-	-	-	-	-	-	Y
App Authenticator	authentic.your.app.authenticator@v1.5	100k+	Y	-	-	-	Y*	Y	-	Y
andOTP	org.shadowice.flocke.andotp@v0.9.0.1-play	100k+	Y	-	-	Y	Y^	-	-	Y
Zoho OneAuth	com.zoho.accounts.oneauth@v2.1.0.5	100k+	-	-	Y*	-		-	-	-
Authenticator Pro	me.jmh.authenticatorpro@v1.15.10	100k+	-	-	-	Y	Y	-	-	-
SAASPASS	com.solidpass.saaspass@v2.2.28	100k+	-	Y	-	-	-	-	-	-
Authentic Password	authentic.password.authenticator.pro@v1.3	100k+	Y	-	-	-	-	Y	-	Y
Mobile Authenticator	authenticator.mobile.authenticator@v1.7	100k+	Y	-	-	-	-	Y	-	Y
	TOTAL apps	**-**	**7**	**3**	**9**	**5**	**6**	**7**	**4**	**9**
	TOTAL installs	**181.5M+**	**101M+**	**6.1M+**	**73.7M+**	**1.8M+**	**1.5M+**	**2M+**	**1.7M+**	**2.2M+**

Table 1: Overview of the backup mechanisms supported in each app. **Y*** indicates that there is a serious security flaw in the implementation and/or usage of cryptography (see Section 5.3). **Y^** indicates support for multiple types of encrypted file exports (see Section 5.3.4). Values in parentheses were obtained from documentation and observation only (see Section 6.4).


Export the vault

This action will export the vault out of Aegis' internal storage. Select the format you'd like your export to be in:

Export format
Aegis (.JSON)

Encrypt the vault

You are about to export an unencrypted copy of your Aegis vault. **This is not recommended.**

I understand the risk

Figure 2: Plaintext warning in the *Aegis Authenticator* app.

5M+ users. Plaintext cloud syncing was also the only backup mechanism supported by the *SAASPASS* app; if enabled, it backed up the TOTP fields in plaintext via the XMPP protocol. Though the feature was optional and off by default, *SAASPASS* regularly prompted the user to enable it. Similarly, the *2FAS* app prompted users to enable remote backups to Google Drive, which were in plaintext by default. The app did support an additional option to encrypt remote backups, but a UX flaw still leaked plaintext in some cases (see Section 5.3.4).

Of the 12 apps that supported plaintext backups, only 4 provided a warning to users that exporting plaintext TOTP backups is risky. *2FAS*, *Authenticator Pro*, *andOTP*, and *Aegis Authenticator* (see Figure 2) each displayed a warning and required the user to verify their intent to export plaintext by clicking a checkbox, toggle, or confirmation button. *Authenticator Pro* displayed a detailed security warning before exporting plaintext to file, but also supported a feature that allowed the user to copy/paste a plaintext backup to the clipboard with no warning.

5.3 Remote Backups *with* Encryption

More than half (15) of the apps we analyzed, comprising over 74 million installs, supported encrypted backups of TOTP data. However, the implementations of many of these apps introduced a range of vulnerabilities, up to and including giving the backup service the ability to decrypt the backup.

All of these apps implemented a similar architecture, in which they encrypted all or part of the TOTP backup using a symmetric encryption key and uploaded/exported the resulting ciphertext to a storage location, such as a third-party cloud service, Google Drive, or user-selected local and remote locations (see Table 1). We discuss the security and privacy impacts of the storage location in Sections 5.5 and 6.1.

All except one of the 15 apps derived the encryption key from a user-provided password. If an attacker obtains the backup, then they can try to crack the ciphertext just like they would a password hash. The feasibility of such offline attacks relies primarily on the strength of the password itself (discussed in Section 5.3.1) and secondarily on the KDF algorithm and how it is configured (discussed in Section 5.3.2).

Microsoft Authenticator was the only app that did not derive keys from user-provided passwords. Instead, it obtained randomly-generated AES-256 keys from a Microsoft key service. Using random keys is cryptographically ideal, but it introduces a significant key management challenge, which we discuss in Section 5.3.3.

The following subsections detail the data in Tables 2 and 3.

5.3.1 Password Policies

Most apps that derived encryption keys from user-provided passwords had severely inadequate password policies, making the encrypted backups vulnerable to trivial guessing techniques implemented in modern password cracking tools (RQ3). It is well established that people typically create weaker passwords on mobile devices [49, 75]. Best practices, such as rejecting weak passwords and nudging users to create stronger passwords, help to mitigate this risk.

The most important aspect of a password policy is the minimum length [38], but most apps accept incredibly short passwords; several apps even accept just a single character (see Table 3). A few apps did employ various levels of block lists. For example, *Duo Mobile* rejected known weak passwords and *Latch* used HaveIBeenPwned[17] to display a warning when the password was included in previous data breaches. It seems likely that many users would consider TOTP backups important, which suggests that password strength meters could nudge them to select stronger passwords [27, 74]; the only app to implement a strength meter was *Aegis*.

By default, *Auth0 Guardian* suggested a random password of length 10, but allowed a user to enter their own password instead. Similarly, *Authenticator* and *App Authenticator* could optionally suggest a password composed of 4 words selected from a list of 6,566 words (10^{15} permutations). Sadly, these apps also used a static salt, which largely negates any benefits of suggesting passwords (see Section 5.3.2).

5.3.2 KDFs and their Configurations

Most apps that we analyzed used a key derivation function (KDF) and/or KDF configuration that was wildly inadequate to meaningfully frustrate offline attackers (RQ3).

Among the apps that derive keys from passwords, *Zoho OneAuth* and *TOTP Authenticator* were at the highest risk to trivial offline attacks because they used a single round of SHA-256 as a KDF. SHA-256 is a prevalent cryptographic hash function that has been widely and heavily optimized to execute quickly in modern hardware. However, fast execution is the antithesis of the design goals for KDFs, which aim to execute slowly. SHA-256 is not a KDF and, therefore, should never be used alone for key derivation.

Authenticator and *App Authenticator* utilized the KDF defined in PKCS#12 [51], which is not appropriately hardened to mitigate offline attacks on modern hardware and has been deprecated in favor of PBKDF2 [50].

PBKDF2 [50] was the most commonly used KDF ($n = 7$) among the apps we analyzed. Internally, PBKDF2 computes a Hash-based Message Authentication Code (HMAC) using a given cryptographic hash function. Of the 7 apps that used PBKDF2, 4 used SHA-1 and 3 used SHA-256. Most KDF configuration recommendations are made in the context of secure password storage, but they still serve as a useful reference point for key derivation. NIST [38] states that PBKDF2 iterations should be as many "as verification server performance will allow," but *at least* 10,000. While all of the apps that we looked at do 10,000 iterations or more, many argue this value is too low for modern usage. OWASP [7] highlights that values should take into account the underlying hash function and recommends 720,000 iterations for PBKDF2-HMAC-SHA1 and 310,000 iterations for PBKDF2-HMAC-SHA256. The highest PBKDF2 iteration count among apps we analyzed was only ~100,000 (in *andOTP* and *LastPass Authenticator*).

Only 3 apps used modern memory-hard KDFs. While PBKDF2 is only CPU-hardened, modern KDFs are designed to also be memory-hardened, which significantly increases the cost of execution. *Yandex.Key* and *Aegis Authenticator* both used scrypt [55], while *Duo Mobile* used argon2i [15]. *Duo Mobile* used the *libsodium* library, which chooses different KDFs, configs, and algorithms depending on the available resources. The configuration for all of these apps exceeds the OWASP recommendations [7] for password storage. In Section 6.2, we argue that the backup mechanisms in TOTP apps should configure KDFs to run significantly more slowly.

While the vast majority of apps that we analyzed correctly utilized random salts when deriving keys from passwords, *Authenticator* and *App Authenticator* both used the hard-coded salt value "ROYALEWITHCHEESEROYALEWITHCHEESE", which makes the TOTP backups generated by these apps vulnerable to rainbow table attacks [69]. Given the uniqueness of this value, we believe that *App Authenticator* is an unauthorized repackaged clone, since it is littered with ads while *Authenticator* is not.

5.3.3 Key Management

The encrypted TOTP backups created by several apps could be decrypted by the remote service storing the backup (RQ3). The security of a cryptographic architecture relies not only on how an encryption key is generated, but how that key is handled and stored. Several apps sent both the ciphertext **and** the encryption key (or password from which it was derived) to the same entity, allowing it to decrypt the TOTP backup.

Microsoft had the technical capability to decrypt the TOTP backups created by *Microsoft Authenticator* because it had access to both the encrypted TOTP backup and the associated key. As mentioned previously, *Microsoft Authenticator* obtained randomly-generated keys from a Microsoft key service instead of deriving keys from passwords. However, the app also stored the encrypted TOTP backup on a Microsoft con-

[17] https://haveibeenpwned.com/

Abbreviated Name	Encrypted?	PII to use cloud backups					Backup Location	TOTP Data Leaked			Obtain Backup With...
		phone	email	name	dob	photo		secret	label	issuer	
Microsoft Authenticator	Yes*	Y	Y	Y	Y	-	activity.windows.com	Y	Y	Y	Microsoft account
Duo Mobile	Yes	-	Y	Y	-	Y	www.googleapis.com	-	Y^	Y^	Google account
Twilio Authy	Yes	Y	Y	-	-	-	api.authy.com	-	Y	Y	SMS only
Latch	No	-	Y	-	-	-	latch.elevenpaths.com	Y	Y	Y	Latch account
LastPass Authenticator	(Yes)	-	Y	-	-	-	(lastpass servers)	(Y)	(Y)	(Y)	Lastpass account
2FAS	No	-	Y	Y	-	Y	www.googleapis.com	Y	Y	Y	Google account
	Yes*	-	Y	Y	-	Y	www.googleapis.com	Y^	Y^	Y^	Google account
Yandex.Key	Yes*	Y	-	-	-	-	registrator.mobile.yandex.net	Y	Y	Y	SMS only
Salesforce Authenticator	Yes*	Y	-	-	-	-	authenticator-api.salesforce.com	Y	Y	Y	SMS only
TOTP Authenticator	Yes*	-	Y	Y	-	Y	www.googleapis.com	Y	Y	Y	Google account
Zoho OneAuth	Yes*	-	Y	-	-	-	accounts.zoho.com	Y	Y	Y	Zoho account
SAASPASS	No	Y	-	-	-	-	104.154.49.147	Y	Y	Y	SMS only

Table 2: Overview of the backup mechanisms that automatically sync data to the cloud. **Yes*** indicates a serious security flaw in the implementation and/or usage of cryptography (see Section 5.3). **Y^** indicates the field is conditionally included in the backup as plaintext (see Section 5.5). Values in parentheses were obtained from documentation and observation only (see Section 6.4).

trolled storage service. This behavior is clearly documented in a Microsoft blog post [76]. Microsoft engineers also acknowledged this design on Twitter,[18] but stated that the company implemented internal security procedures to reduce the risk of attack.[19] On the iOS version of the *Microsoft Authenticator* app, the ciphertext is stored in the user's iCloud, removing Microsoft's technical capability to decrypt the TOTP backup since it only has access to the key [76]. It seems that the Android app could achieve a split-knowledge architecture like the iOS app by leveraging the Android Auto Backup system, or Google Drive directly, to store encrypted backups.

Several apps that derive keys from passwords also ran into the same vulnerability, but in less transparent ways. Each of *Yandex.Key*, *Zoho OneAuth*, and *Salesforce Authenticator* sent the backup password and the encrypted backup to domains controlled by each of those apps' developers, giving them the technical capability to decrypt the backups. The *Yandex.Key* app used Yandex servers to perform a password strength test, which should happen locally on the device instead. In its documentation, *Zoho OneAuth* states that *"The reason for [encrypting backups] is to make sure that your OTP secrets are stored securely and not accessed by anyone (including Zoho). **You should note that only the encrypted secrets will be stored by Zoho and not the passphrase**"* (bold theirs) [81]. It is unclear whether the bold text is meant to disclose that the password is sent to Zoho servers, but it certainly does not explain *why* it is transmitted. Regardless, there is no way for users to verify that Zoho does not, in fact, store the passphrase on the server. The *Zoho OneAuth* backup implementation seems to flagrantly violate the confidentiality goals defined in the documentation because Zoho has the technical capability to decrypt the TOTP backups. *Salesforce Authenticator* sent the password to Salesforce servers so that it could be used "to verify [users'] ownership of the backed-up accounts" [65] during recovery. Since *Salesforce Authenticator* also required an SMS OTP during authentication, the short recovery PIN does not provide enough additional security to warrant allowing Salesforce to decrypt the TOTP backup.

The *TOTP Authenticator* app suffers from a different key management issue: hard-coded keys. Backups uploaded to Google Drive are encrypted using a key derived from a static, hard-coded password, which is equivalent to hard-coding the key directly in the app source code. Anyone who decompiles the app can obtain the key, granting attackers with access to the backup the ability to instantly decrypt it. Surprisingly, the app does derive a key from a user provided password when encrypting TOTP backups exported to a file. The app should leverage the same behavior when backing up to Google Drive.

5.3.4 How TOTP Backups are Encrypted

Of the 15 apps using encrypted backups, 5 used modern Authenticated Encryption with Associated Data (AEAD) primitives, while others used deterministic encryption (AES-ECB) and many did not provide integrity over the ciphertext (RQ3).

AEADs As seen in Table 3, 5 apps secured backups with an AEAD, primitives that encrypt and authenticate messages in one atomic API call, thus reducing the chance of developer error compared to employing encryption and Message Authentication Code (MAC) primitives individually. Xsalsa20_Poly1305 was used by 2 apps (via the `libsodium` [14] secret_box API), while 3 others used AES-GCM. Next, we discuss the 8 apps[20] that used AES modes of operation that only provide confidentiality.

AES-CBC Of the 5 apps that use AES-CBC, 4 correctly generated a random initialization vector (IV), but the *TOTP Authenticator* app used an IV of all zeros. This flaw allows attackers to determine whether multiple backups start with the same plaintext blocks, but the real world impact is arguably immaterial. As mentioned in Section 5.3.2, this app uses a

[18] https://web.archive.org/web/20221003155439/https://twitter.com/Alex_T_Weinert/status/1195841144304758786

[19] https://web.archive.org/web/20221003155253/https://twitter.com/Alex_T_Weinert/status/1195841594814976000

[20] Some apps used AES with unknown modes of operation (see Table 3).

Abbreviated Name	Key Source	Password Min Len	KDF and Configuration	KDF Salt	Encryption Algorithm	Ciphertext Integrity	Decryption Heuristic
Microsoft Authenticator	Random*	n/a	n/a	n/a	AES-128-CBC	HMAC-SHA256	n/a
Zoho OneAuth	Password*	3	SHA-256 i = 1	none	AES-256-ECB	none	Base32
Salesforce Authenticator	Password*	4	PBKDF2-HMAC-SHA256 i = 10,000	random	AES-256-CBC	none	JSON
Yandex.Key	Password*	6	scrypt N = 2^15, r = 20, p = 1	random	Xsalsa20_Poly1305	AEAD	n/a
TOTP Authenticator	Password	8	SHA-256 i = 1	none	AES-256-CBC	none	JSON
Authenticator	Password	10	PKCS12-SHA256 i = 65,536	hard coded	AES-256-ECB	none	URI
App Authenticator	Password	10	PKCS12-SHA256 i = 65,536	hard coded	AES-256-ECB	none	URI
Auth0 Guardian	Password	1	(PBKDF2-HMAC-SHA1) (i = 10,000)	(random)	(AES-256)	(HMAC)	(n/a)
Authenticator Pro	Password	1	PBKDF2-HMAC-SHA1 i = 64,000	random	AES-256-CBC	none	JSON
2FAS	Password OpenPGP	1	PBKDF2-HMAC-SHA256 i = 10,000	random	AES-256-GCM	AEAD	n/a
Aegis Authenticator	Password	2	scrypt N = 2^15, r = 8, p = 1	random	AES-256-GCM	AEAD	n/a
andOTP	Password	4	PBKDF2-HMAC-SHA1 i = [140,000 - 160,000]	random	AES-256-GCM	AEAD	n/a
Twilio Authy	Password	6	PBKDF2-HMAC-SHA1 i = 10,000	random	AES-256-CBC	none	Base32
Duo Mobile	Password	10	argon2i m = 128 Mb, t = 6, p = 1	random	Xsalsa20_Poly1305	AEAD	n/a
LastPass Authenticator	Password	12	(PBKDF2-HMAC-SHA256) (i = 100,100)	(random)	(AES-256)	(HMAC)	(n/a)

Table 3: Cryptographic details of app backup mechanisms. The asterisk (*) indicates that the app leaks the encryption key and/or password to the same service which stores the ciphertext, allowing that service to decrypt the TOTP backup (see Section 5.3.3). Square brackets indicate the min and max of a range, inclusive. Values in parentheses were obtained from documentation and observation only (see Section 6.4). The abbreviations for KDF configurations are: SHA/PKCS12/PBKDF2 (i = iterations), scrypt (N= CPU/memory cost, r = block size, p = parallelism), and Argon2 (m = memory, t = time/iterations, p = parallelism).

seriously flawed process to derive keys from user-provided passwords. Attackers will undoubtedly crack the password (and thus the key) directly, rather than analyze the ciphertext.

As mentioned in Section 5.2, the *2FAS* app does allow users to enable a feature that automatically encrypts and uploads TOTP backups to the user's Google Drive. However, the app did not allow the user to enter a backup password *before* enabling the Google Drive backup mechanism, which resulted in existing TOTP data being uploaded to Google Drive in plaintext. Once a password was provided, all existing and future TOTP accounts were encrypted with AES-CBC using a random IV before they were backed up to Google Drive.

AES-ECB Alarmingly, 3 of the apps encrypted TOTP backups using AES-ECB, which is deterministic and does not provide indistinguishability under chosen plaintext attack (IND-CPA), commonly referred to as semantic security. Ironically, the use of AES-ECB in these particular apps is basically immaterial because they each have other serious security flaws that allow trivial decryption of the TOTP backups. *Zoho OneAuth* uses AES-ECB, but Zoho servers already have the technical capability to decrypt backups (see Section 5.3.3). The *Authenticator* and *App Authenticator* apps also use AES-ECB, but backups in these apps are already vulnerable to rainbow table attacks (see Section 5.3.2). Similar to the misuse of IVs in AES-CBC mode discussed previously, attackers are most likely to attack the backup passwords directly than hunt for clues in the ciphertext output by AES-ECB. Still, these apps should abandon AES-ECB and use an AEAD instead.

Integrity *Microsoft Authenticator* was the only app not to use an AEAD for encryption, while providing integrity over the ciphertext (using HMAC-SHA256). All other apps that used AES-CBC and AES-ECB had no cryptographic mechanism to authenticate the ciphertext before performing decryption. Instead, these 7 apps relied on a range of heuristics to determine whether the decryption succeeded. Checking whether the decrypted plaintext was a valid JSON object or URI were common techniques used by 5 apps. Interestingly, the *Twilio Authy* and *Zoho OneAuth* apps only encrypted the Base32 encoding of the TOTP secret. The random secret would normally prevent identifying whether the correct decryption key was used, but the Base32 encoding provides a reliable heuristic (see Appendix A). Offline attackers can use the same heuristic, so not including a MAC for CBC or ECB mode does not enhance security and seems to be an oversight.

Asymmetric Encryption *andOTP* was the only app that supported backups with asymmetric encryption. The app could send the plaintext backup to a third party app,[21] which encrypted the data using a PGP key and returned the resulting ciphertext. There is a clear risk of compromise when sending data to a third-party app. To reduce the attack surface, PGP functionality should be implemented directly within the app using trusted libraries.

5.4 Android Auto Backup

Android 6.0 and above supports a backup system that automatically uploads an app's data to the user's Google Drive [12]. Android apps are opted into Android Auto Backup (AAB) by default, but developers can explicitly opt-out by setting `android:allowBackup="false"` in the app's manifest file. The official docs suggest that developers opt-out "...if [the] app deals with sensitive information that Android shouldn't back up." [12] We argue that TOTP fields, especially the secret, are sensitive and should not be backed up via AAB without additional protections.

The AAB documentation [12] states that backup data uploaded to Google Drive is "...end-to-end encrypted on devices running Android 9 or higher using the device's pin, pattern, or password." We did not review the Android source code to verify implementation details, but it is apparent that encryption is used to protect app data. The use of end-to-end encryption is important to protect the backups at rest in Google's data centers from internal attackers, but the device's pin/pattern/password is likely to be low entropy and may not provide any meaningful protection against offline attacks.

Just over half (12) of the apps in our dataset explicitly opt out of AAB. We manually triggered the AAB functionality for each of the 10 apps that supported it and observed which data, if any, was restored after uninstalling and reinstalling the app. We could not get AAB to run without error for 3 apps that supported it.[22] While we could not observe the behavior for these apps, we included them in our analysis because AAB may work on Android versions that we did not test. *Google Authenticator* did support AAB, but did not backup any of the TOTP fields via this mechanism; it only backed up application settings, such as Dark Mode preferences.

Each of the remaining 6 apps whose AAB behavior we could observe backed up all of the TOTP fields (secret, issuer, and label) via AAB. The user's Google account was a single point of failure for users of the 5 apps that relied solely on the security protections native to AAB. An attacker with access to the Google account could read the TOTP backup and generate valid OTPs for the user's accounts.

[21] https://play.google.com/store/apps/details?id=org.sufficientlysecure.keychain

[22] *Aegis Authenticator*, *andOTP*, and *FreeOTP Authenticator*

The *Auth0 Guardian* app was the only app whose AAB behavior we could observe that added additional protections before the TOTP backups were sent to Google Drive via AAB. The app required the user to enter a backup password, which was used to enable the native encryption support in the Realm database [2] that the app used to store TOTP data and other app settings (see Table 3). We believe that the entire encrypted Realm database was backed up via AAB because the Auth0 app also required the backup password upon recovery.

It is also worth noting that even though we could not observe their AAB behavior, *andOTP* and *Aegis Authenticator* were the only apps that allowed the user to opt in/out of using AAB. Both apps required a backup password if AAB was enabled, which, we believe, would be used to derive a key and encrypt the TOTP backup before sending it to Google Drive.

5.5 Privacy Implications

This section explains how the backup mechanisms in TOTP apps can leak personal user information to third parties (RQ1).

PII to Use App Only 2 apps required users to provide personal information to the app developer in order to use the app at all, even if the backup mechanism was not enabled. *Twilio Authy* required the user to enter an email address and prove control of a phone number via SMS OTP or voice call. *Latch* required the user to create an account in order to use the app, which required an email address.

PII to Enable Backups Apps that supported local file exports and sharing did not request any personally identifiable information (PII) to use those features, but the use of plaintext exports (see Section 5.2) can have an obvious impact on users' privacy depending where they are sent and stored.

The ability to automatically sync backups to the cloud universally required users to divulge at least some PII so that they could be authenticated during recovery (see Table 2).

Of the 11 apps that supported a cloud sync backup mechanism, 8 required the user's email address and 5 required the user's phone number to utilize this feature. The *Microsoft Authenticator* app required a Microsoft account to enable cloud sync, which required the most PII of any app that we analyzed: phone number, email address, name, and date of birth. *Zoho OneAuth* also required account creation for cloud sync and collected user email address, country, and state.

The 3 apps that automatically sync backups to the user's Google Drive (i.e., separately from the aforementioned apps using AAB) used OAuth/OpenID to perform this upload on the user's behalf. Doing so granted each of those apps permission to read the user's primary email address, name, and account photo. Though not technically required for the backup functionality to work, the user's name and account photo fields are included in the narrowest set of permissions that developers can request.

Leaking TOTP Labels and Issuers Section 5.3 discussed app developers who have the technical capability to decrypt TOTP backups and read their content, including secret, issuer, and label. Here, we discuss TOTP data that is leaked directly to third parties in plaintext. Several apps that supported encrypted cloud sync only encrypted the TOTP secret and included all other TOTP fields in backups as plaintext.

Interestingly, *Duo Mobile* conditionally includes either the TOTP issuer or TOTP label in plaintext depending on whether the website/service is on an internal list of popular websites/services. If the website/service is a member of the list, then the TOTP issuer is included in plaintext, while the TOTP label is included in plaintext if the website/service is "custom" (i.e., not on the list). Both *Zoho OneAuth* and *Twilio Authy* only encrypted the secret, which meant the issuer and label are sent to the Zoho and Twilio servers in plaintext.

Functionally, there is no reason to avoid encrypting the TOTP issuer and label fields in the TOTP backups and it is not immediately obvious how this decision improves the user experience. These plaintext fields appear to have no impact whatsoever on the UX of the *Zoho OneAuth* app; no TOTP data is displayed during recovery unless the correct backup password is entered. In the *Duo Mobile* and *Twilio Authy* apps, however, the TOTP issuer and label are displayed in the app even if the backup password is not entered at all, or is incorrect. At best, one could learn which accounts they may be locked out of if they cannot recall their backup password. While backups generated by *Duo Mobile* are stored on Google Drive, the *Twilio Authy* app and *Zoho OneAuth* app each store backups on their own servers. This means that any user of *Twilio Authy* or *Zoho OneAuth* who enables cloud backups is unknowingly sending those companies the names of the websites/services they use and the usernames for their accounts on those platforms.

5.6 Reliance on Passwords, SMS, and Email

Of the 19 apps that supported automatically uploading TOTP backups to the cloud (via cloud sync features and/or AAB), 15 required the user to authenticate to the cloud service storing the backup, while 4 relied solely on SMS OTP to authenticate users during recovery (see Tables 1 and 2) (RQ2).

The *SAASPASS* app supported a dangerous combination of uploading plaintext TOTP backups to the cloud and relying solely on SMS OTP to authenticate users during recovery. Any attacker who leverages any of the well-known and numerous techniques to hijack the user's phone number will gain immediate access to the full TOTP backup, including the TOTP secrets. The app does allow users to optionally enforce a 20-hour delay on sending the recovery OTP via SMS, which can give the user critical time to switch to a different TOTP app while rotating their TOTP secrets and/or regain control of their phone number. It is unclear how many users discover this option in the security menu and enable it in practice.

By default, each of *Twilio Authy*, *Yandex.Key*, and *Salesforce Authenticator* also relied solely on SMS OTP to authenticate users during recovery, but did encrypt TOTP backups using a key derived from a password before uploading them to the cloud. To compromise the backup, an attacker who hijacks the phone number will still need to conduct an offline attack to guess the backup password. Presumably, many users will realize that their phone number has been compromised once their phone stops working [20] and take action. Thus begins a race. In addition to regaining control of their phone number, the user should begin rotating their secrets on each individual account protected by TOTP. The question is whether they can rotate everything before the attacker successfully cracks the TOTP backup, enabling them to generate valid OTPs and attempt to log into their accounts. If the backup password could be cracked quickly, then chances are high that it was relatively weak, which is unsurprising considering the password policy issues discussed in Section 5.3.1. Given the fact that password reuse is rampant [10], it seems likely that the user may also have weak account passwords, which may allow the attacker who hijacked the user's phone number to fully compromise their online accounts protected by TOTP 2FA.

The remaining 15 apps that supported remote cloud backups required the user to log into to an account on the cloud service to obtain the TOTP backup. This begs the question: *What are the authentication mechanisms for those accounts?*

To use the *Latch* app at all, we were required to create an account on the Latch website, which only required a username and password. There were no indications of any support for 2FA. During recovery, the backup could be obtained with only the username and password.

Cloud backups on *Microsoft Authenticator* required creating a Microsoft account. In addition to a username and password, creating this account required proving control of a phone number and providing an email address. To obtain the backup during recovery, we were required to prove control of the phone number again (i.e., SMS 2FA).

LastPass Authenticator actually relied on the LastPass Password Manager app, developed by the same company, to encrypt and store TOTP backups. The password manager is free to use, but requires creating an account by providing an email and password. The *LastPass Authenticator* app then requires that at least one 2FA option is enabled on the password manager account. However, the only 2FA options freely available on LastPass accounts that did not rely on possession of a device was printable recovery codes. Of course, if the user enables TOTP 2FA or Push 2FA and only has a single device, then they will not be able to log into their LastPass account in the event that they lose their phone.

A Zoho account was required to enable cloud backups in *Zoho OneAuth*, which required an email and password. Zoho does support multiple methods of 2FA [4], including SMS 2FA, but none were required.

As mentioned previously, the user's Google account is commonly used to store remote backups; apps upload TOTP backups to Google Drive directly, or via Android Auto Backup. In an effort to combat low adoption rates and increase the security of accounts, Google announced in 2021 that it would begin requiring hundreds of millions of users to enable 2FA on their accounts [48]. Arguably, this decision reflects a larger, industry-wide paradigm shift [11] and unquestionably improves the overall security posture of account security. However, at the time of our analysis, the workflow to enable 2FA on a Google account required users to initially choose one of three specific methods: Push 2FA, SMS 2FA, or a security key. Google does support enabling several additional 2FA methods after initial setup [1], but it seems almost certain that the vast majority of users will stick with either SMS 2FA or Push 2FA given the general lack of knowledge about 2FA and its security benefits [22, 41, 62], low organic adoption rates [36], that adoption of security keys is quite rare among the general population [9], and that these alternative 2FA methods are completely optional. If SMS 2FA is enabled, then the Google account is protected by the same exact technologies that users of TOTP are likely trying to avoid: passwords and SMS. If Push 2FA is enabled, then the user could face account lockout across all of their online accounts protected by TOTP 2FA if their device is inaccessible; they will be unable to generate OTPs because the device on which the TOTP secrets were stored is lost and, at the same time, they will be unable to recover those TOTP secrets because they were backed up to their Google account, which requires them to approve a notification sent to the lost device to log in. Registering multiple personal devices for Push 2FA can help recovery if only a single device is lost, but many users only have a single device.

For cloud-based backup mechanisms, this account recovery conundrum inevitably reduces the security of the TOTP 2FA scheme to just another layer of the same authentication mechanisms that TOTP 2FA is meant to supersede: username/password, SMS, and/or email (RQ2).

6 Discussion

In this section, we discuss the dangers of plaintext backups, make recommendations, describe our responsible disclosure, and discuss the limitations of this study.

6.1 The Dangers of Plaintext Backups

The dangers of plaintext backups are largely self-evident, but warrant some unpacking to gauge the real-world risk.

Universally among the apps we analyzed, an attacker with access to the plaintext TOTP secret also learned the names of the websites/services on which the user had accounts and/or the usernames for those accounts, allowing them to leverage classic password attacks. Plaintext TOTP backups grossly undermine the security benefits of TOTP 2FA.

The majority of apps that support plaintext backups do so via file export or sharing. File exports from a TOTP app will not help a user recover if the export is stored locally on the lost device. Sending plaintext backups remotely can leak the TOTP data to every actor involved in transporting and storing the backup. For example, if a user sends the plaintext file via email, then it is stored in the sender's outbox and the recipient's inbox. Some users may understand these risks and choose a secure alternative, such as sending the backup to another personal device via an end-to-end encrypted chat app, such as Signal.[23] However, we expect that many users will not grasp the sensitivity of plaintext backups since so few apps warned users about the associated risks (see Section 5.2).

One use case in which plaintext exports are particularly useful is migration between apps. Many apps supported importing the plaintext backups from other TOTP apps. As long as the plaintext export is deleted after the local migration is complete, there should be little to no risk of compromise.

Google Drive is a common storage location for TOTP backups; many apps store backups there via Android Auto Backup, several apps upload there directly using SDKs, and many Android users will likely choose it as a storage location via the sharing menu. Storing plaintext TOTP backups in Google Drive simply outsources responsibility for securing them to Google. Many users may, in fact, have a threat model in which this is a perfectly reasonable backup solution given Google's existing security mechanisms. Google Drive is also operated by a separate company than all of the apps that use it for storage, potentially allowing the TOTP backups to hide among the masses. We argue that these assumptions do not hold for *Latch* and *SAASPASS* because their employees know that their servers are storing plaintext TOTP backups, which increases the risk of internal attacks. The security of remotely-stored plaintext backups against external attackers is directly dependent on the authentication mechanisms protecting that account. An attacker who compromises a user's *Latch* or Google accounts can simply install the relevant TOTP app to automatically restore **all** of the user's TOTP secrets.

6.2 Recommendations

According to the Google Play Store, the TOTP apps that we analyzed have a collective install count of over 180 million (see Table 1). The following recommendations will help app developers improve their backup and recovery mechanisms to address security vulnerabilities and respect user privacy.

Apps should consider not supporting plaintext backups. If they are supported, then users should be clearly warned about the associated risks.

Including any TOTP fields in backups as plaintext violates user privacy. Apps should encrypt all TOTP fields, including the secret, issuer, and label. Apps that rely on remote key servers to generate/store random keys should choose different

[23] https://signal.org/

entities for storage of keys and ciphertext; if both are stored with a single entity, then they can decrypt the TOTP backups.

We have several recommendations for apps that derive keys from passwords. First, they should implement well-established best practices to encourage users to create strong passwords (see Section 5.3.1). Second, **NEVER** allow the backup password to leave the app. Once the key is derived from the password, the password should be wiped from memory and the key should be securely stored on the device using the Android Key Store [24] so that it can be used to encrypt TOTP accounts added in the future.

Finally, we recommend that TOTP apps that derive keys from passwords should capitalize on the fact that the KDF operation happens so infrequently and configure it to run significantly slower than existing recommendations for password storage. In contrast to account authentication and unlocking password managers, which can easily happen many times per day, TOTP apps should only ever perform key derivation two times: when the user first enables the backup mechanism, and when the user is attempting to recover. In fact, this is the exact architecture that several apps already implement (e.g., *Duo Mobile*). The original scrypt paper [55] asserted that 5 seconds is a reasonable amount of time to wait for file encryption. While that value is subjective, Egelman et al. [26] did find that people are willing to endure longer delays in security contexts when they are informed of the threat model and how the delay enhances their security. Given that TOTP apps already show a 30 second countdown[25] indicating when the OTP will be regenerated, we propose that TOTP apps should reuse this familiar UX and dynamically calculate the KDF configuration such that it takes 30 seconds to execute. While the KDF executes, the visual countdown should be displayed along with an explanation of how the delay increases defense against offline attacks and that it will only occur again during recovery. It is likely that the willingness of users to wait has an upper bound regardless of explanation. We believe 30 seconds is a reasonable starting point, but future work should explore this limit empirically.

TOTP apps that derive keys from passwords should adopt the Argon2 KDF because it allows developers to independently configure memory and CPU parameters. As a result, all devices should be able to extrapolate from a small set of tests a KDF configuration that should execute within the desired clock time. Encouragingly, the *andOTP* app did dynamically calculate the number of rounds of PBKDF2 required to keep the clock time within 1 second, but only used this technique when encrypting the TOTP backup in Android Auto Backup (see Section 5.4) and not when backing up to Google Drive. Higher end devices could consider the historical trends in device specs (e.g., the median amount of memory of phones sold within the last 2 years) to ensure that recovery can run without error on lower end devices, even if it takes longer than 30 seconds. For example, a user who loses their brand new phone would reasonably expect to be able to recover on an older device running the same TOTP app.

[24] https://developer.android.com/training/articles/keystore

[25] While most TOTP apps are capable of using any time window defined by the server, 30 seconds is almost universally used in practice.

6.3 Responsible Disclosure

Here, we outline our best efforts to disclose substantive issues to the respective app developers and summarize the responses we received as of November 4, 2022[26]. We felt there was nothing to disclose for the following 6 apps: *Google Authenticator*, *LastPass Authenticator*, *FreeOTP Authenticator*, *Authenticator Pro*, *Aegis Authenticator*, and *Auth0 Guardian*. The developer of *andOTP* announced its deprecation [52] after our analysis, so no report was filed. *TOTP Authenticator* did not respond to our email and Twitter communications asking for a security contact, so no report was filed. We contacted each of the remaining 14 app developers and gave them at least 90 days to respond to our report.

The issues in the *Twilio Authy* app that we disclosed to Twilio in 2020 [35] (v24.3.1) persisted in the app over 2 years later (v24.8.5). Twilio stated in October 2022 that they are "committing to increase the length of the Backup password" and "significantly increasing the number of [PBKDF2] iterations." In response to our disclosure that the Authy backup mechanism sends the TOTP issuer (i.e., website/service name) and label (i.e., the account username) to Twilio servers in plaintext, Twilio stated that users are able to set the issuer field to any custom value and that these fields are required "...as an aid to help the users to know what tokens they encrypted in multi-device scenarios." We find it extremely unlikely that many users change the default value of the TOTP issuer, which is almost universally set as their username by the website on which they are enabling TOTP, and wanted to know the percentage of users who take this action in practice. However, Twilio claimed that they do not track changes to the TOTP issuer field. Twilio stated that they are considering updates to their privacy policy. Twilio uses BugCrowd, but we chose not to disclose via that platform because "Twilio does not permit public disclosure at this point in time."[27]

Latch did not rule out our suggestion of allowing users to opt in/out of TOTP backups, but rejected our strong recommendation to implement end-to-end encryption, stating that the current behavior of sending TOTP backups to Latch servers in plaintext was "by design" and "intended."

Microsoft confirmed the backup mechanism in their Android app was "by design" and highlighted their internal security mechanisms. They did not respond to our suggestion to store ciphertext in Google Drive instead of a Microsoft storage service, nor our inquiry about why a more robust backup mechanism was implemented on the iOS version of the app.

[26] See Section 7 for supplemental material available online.

[27] https://bugcrowd.com/twilio

Duo confirmed our report and pointed us to the Duo Privacy Data Sheet [13], which they said listed Google as a sub-processor and disclosed the collection of username/email. Regarding their backup design, Duo stated, "[b]y allowing users to see their accounts in a non-restored state, Duo's goal is to help facilitate them setting up their accounts with their services (e.g. Amazon) more easily. They do not have to recall every service they set up OTP accounts for on their own."

Salesforce responded to our disclosure report by linking to two support articles [65, 66] on backup and recovery. Each of these articles stated "...your encrypted TOTP data is stored on Salesforce servers... During backup and restore events, your passcode is used to verify your ownership of the backed-up accounts." Neither the documentation nor the response from Salesforce addressed the fact that Salesforce has the technical capability to decrypt TOTP backups (see Section 5.3.3).

The developer of *Authenticator* was very receptive to our disclosure and released an updated version of the app (v1.2.4) that (1) switched from static to random salts; (2) switched from AES-ECB to AES-CBC; (3) set minimum password length to 20; and (4) warned of plaintext export risks.

The developer of *Code Generator* said they would update the app when they had "...enough available time and resources..." Communication with *2FAS* ceased after a developer asked for our PGP key, which we provided.

At the time of publication, the remaining 6 app developers to whom we disclosed our findings never replied. Discussions are on-going with several companies.

6.4 Limitations

We focused on Android only, so future work should analyze the behavior of TOTP 2FA apps on iOS. We expect that most apps will exhibit the same behavior on both platforms, but we know this is not the case for all apps (e.g., *Microsoft Authenticator* discussed in Section 5.3.3).

As we discuss in Section 5.4, we could not get Android Auto Backup to run without error for several apps.

We were unable to verify the cryptographic primitives used by the *LastPass Authenticator* and *Auth0 Guardian* apps. The *LastPass Authenticator* app actually relies on the LastPass Password Manager app[28] (developed by the same company) to enable encrypted cloud sync for TOTP backups. We found documentation that it derives keys from the master password using 100,100 rounds of PBKDF2 with random salts, calculates an authentication hash, and encrypts data using AES-256 [3, 6]. We observed in the decompiled code that *Auth0 Guardian* uses RealmDB to store application data, which has native support for encrypting data using AES-256 for confidentiality and an HMAC for integrity [2]. As appropriate, the value for these two apps in Tables 1, 2, and 3 are clearly marked as "from documentation and observation only" where we were not able to verify the claims.

[28] https://play.google.com/store/apps/details/?id=com.lastpass.lpandroid

7 Supplementary Materials

We strove to make our work fully verifiable and reproducible. To that end, there are numerous supplementary materials available online at https://allthingsauth.com/totp-apps.

Acknowledgments

This research received funding from the Center for Long-Term Cybersecurity (CLTC) at UC Berkeley, NSF grant CNS-1817249, and NSA contract H98230-18-D-0006. Special thanks to Primal Wijesekera for guidance on the tooling used to capture network traffic; Nathan Malkin for invaluable feedback on early drafts of this paper; and Jason Chan for help analyzing the Microsoft Authenticator app.

References

[1] 2FA Directory. https://web.archive.org/web/20220605042155/https://2fa.directory/int/. (Accessed on 06/05/2022). 1, 12

[2] Encrypt a Realm - Java SDK — MongoDB Realm. https://web.archive.org/web/20220525164833/https://www.mongodb.com/docs/realm/sdk/java/advanced-guides/encryption/. (Accessed on 05/25/2022). 10, 14

[3] How to Use LastPass Password Manager. https://web.archive.org/web/20220606190625/https://www.lastpass.com/how-lastpass-works. (Accessed on 06/06/2022). 14

[4] Introduction to multi-factor authentication (MFA). https://web.archive.org/web/20220607231957/https://help.zoho.com/portal/en/kb/accounts/multi-factor-authentication/articles/mfa-introduction. (Accessed on 06/07/2022). 11

[5] Key Uri Format. Available: https://github.com/google/google-authenticator/wiki/Key-Uri-Format. [Online; accessed: 12-May-2020]. 2, 4, 5

[6] Our Zero-Knowledge Security Model. https://web.archive.org/web/20220606190618/https://www.lastpass.com/security/zero-knowledge-security. (Accessed on 06/06/2022). 14

[7] Password Storage - OWASP Cheat Sheet Series. https://web.archive.org/web/20220530233607/https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html. (Accessed on 05/30/2022). 7

[8] TOTP: Time-Based One-Time Password Algorithm. https://tools.ietf.org/html/rfc6238. [Online; accessed: 02-Oct-2019]. 1, 2

[9] FIDO Alliance and the Path to the Post-Password World. https://media.fidoalliance.org/wp-content/uploads/2020/05/FIDO-Consumer-Research-Report.pdf, May 2020. (Accessed on 06/07/2022). 12

[10] Psychology of Passwords: How Password Hygiene Reduces Your Password Security Risk. https://web.archive.org/web/20220607215159/https://www.lastpass.com/resources/ebook/psychology-of-passwords-2020, 2020. (Accessed on 06/07/2022). 11

[11] Announcement of the Future Requirement to Enable Multi-Factor Authentication (MFA). https://help.salesforce.com/s/articleView?id=000356005&type=1, March 2021. (Accessed on 06/07/2022). 12

[12] Back up user data with Auto Backup. https://web.archive.org/web/20220421053001/https://developer.android.com/guide/topics/data/autobackup, March 2022. (Accessed on 05/25/2022). 10

[13] Duo privacy data sheet. https://web.archive.org/web/20221004030758/https://trustportal.cisco.com/c/dam/r/ctp/docs/privacydatasheet/security/cisco-duo-privacy-data-sheet.pdf, August 2022. (Accessed on 10/03/2022). 14

[14] Introduction - libsodium. https://web.archive.org/web/20220531235007/https://doc.libsodium.org/, March 2022. (Accessed on 05/31/2022). 8

[15] A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: new generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302. IEEE, 2016. 7

[16] J. Bonneau, E. Bursztein, I. Caron, R. Jackson, and M. Williamson. Secrets, lies, and account recovery: Lessons from the use of personal knowledge questions at google. In *Proceedings of the 24th international conference on world wide web*, pages 141–150, 2015. 2

[17] 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. In *IEEE Symposium on Security and Privacy*, pages 553–567, 2012. 1

[18] L. Cavallaro, P. Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Proc. of DIMVA*, pages 143–163. Springer-Verlag, 2008. 3

[19] S. Ciolino, S. Parkin, and P. Dunphy. Of Two Minds about Two-Factor: Understanding Everyday FIDO U2F Usability through Device Comparison and Experience Sampling. In *15th Symposium on Usable Privacy and Security (SOUPS 2019)*, pages 339–356, 2019. 2

[20] Lorrie Cranor. Your mobile phone account could be hijacked by an identity thief. https://web.archive.org/web/20220310162636/https://www.ftc.gov/news-events/blogs/techftc/2016/06/your-mobile-phone-account-could-be-hijacked-identity-thief, July 7 2016. 3, 11

[21] S. Das, A. Dingman, and L J. Camp. Why Johnny doesn't use two factor a two-phase usability study of the FIDO U2F security key. In *International Conference on Financial Cryptography and Data Security*, pages 160–179. Springer, 2018. 2

[22] S. Das, A. Kim, B. Jelen, J. Streiff, L J. Camp, and L. Huber. Towards Implementing Inclusive Authentication Technologies for Older Adults. In *Who Are You?! Adventures in Authentication Workshop*, WAY '19, pages 1–5, Santa Clara, California, USA, August 2019. 12

[23] S. Das, B. Wang, and L J. Camp. MFA is a Waste of Time! Understanding Negative Connotation Towards MFA Applications via User Generated Content. In *Proceedings of the 13th International Symposium on Human Aspects of Information Security & Assurance (HAISA 2019)*, 2019. 2

[24] P. Doerfler, K. Thomas, M. Marincenko, J. Ranieri, Y. Jiang, A. Moscicki, and D. McCoy. Evaluating Login Challenges as a Defense Against Account Takeover. In *The World Wide Web Conference*, pages 372–382. ACM, 2019. 1, 2

[25] A. Drozhzhin. Sms-based two-factor authentication is not safe – consider these alternative 2fa methods instead. Kaspersky Daily, October 16 2018. https://usa.kaspersky.com/blog/2fa-practical-guide/16398/. 1, 2

[26] S. Egelman, D. Molnar, N. Christin, A. Acquisti, C. Herley, and S. Krishnamurthi. Please continue to hold. In *9th Workshop on the Economics of Information Security*, 2010. 13

[27] S. Egelman, A. Sotirakopoulos, I. Muslukhov, K. Beznosov, and C. Herley. Does my password go up to eleven? The impact of password meters on password selection. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2379–2388, 2013. 7

[28] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of the 9th USENIX conference on Operating systems design and implementation (OSDI)*, pages 393–407, 2010. 3

[29] F. M Farke, L. Lorenz, T. Schnitzler, P. Markert, and M. Dürmuth. "You still use the password after all"–Exploring FIDO2 Security Keys in a Small Company. In *16th Symposium on Usable Privacy and Security (SOUPS 2020)*, pages 19–35, 2020. 2

[30] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: user attention, comprehension, and behavior. In *Proceedings of the 8th Symposium on Usable Privacy and Security*, SOUPS '12, New York, NY, USA, 2012. ACM. 3

[31] FIDO Alliance. FIDO2: WebAuthn & CTAP. https://fidoalliance.org/fido2/. (Accessed on 06/06/2022). 2

[32] D. Florencio and C. Herley. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web*, pages 657–666. ACM, 2007. 1

[33] A. Forget, S. Chiasson, and R. Biddle. Helping users create better passwords: Is this the right approach? In *Proceedings of the 3rd Symposium on Usable Privacy and Security*, pages 151–152. ACM, 2007. 1

[34] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *Proc. of the 5th international conference on Trust and Trustworthy Computing (TRUST)*, pages 291–307. Springer-Verlag, 2012. 3

[35] C. Gilsenan, N. Alomar, and S. Egelman. On Conducting Systematic Security and Privacy Analyses of TOTP 2FA Apps. In *Who Are You?! Adventures in Authentication Workshop*, WAY '20, pages 1–6, August 2020. 3, 13

[36] Conor Gilsenan. 2FA Stats. https://allthingsauth.com/2fastats. (Accessed on 06/07/2022). 12

[37] M. I. Gordon, D. Kim, J. Perkins, Gilhamy, N. Nguyenz, and M. Rinard. Information-Flow Analysis of Android Applications in DroidSafe. In *Proc. of NDSS Symposium*, 2015. 3

[38] P. Grassi, E. Newton, R. Perlner, A. Regenscheid, W. Burr, J. Richer, N. Lefkovitz, J. Danker, Y. Choong, K. Greene, and M. Theofanos. Digital identity guidelines: Authentication and lifecycle management, 2017. 7

[39] D. He, T. Katz, and C. Brand. Introducing portability of Google Authenticator 2SV codes across Android devices. https://web.archive.org/web/20210613033604/https://security.googleblog.com/2020/05/introducing-portability-of-google.html. (Accessed on 05/26/2022). 5

[40] Cormac Herley and Paul Van Oorschot. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy*, 10(1):28–36, 2011. 1

[41] I. Ion, R. Reeder, and S. Consolvo. "...No One Can Hack My Mind": Comparing Expert and Non-Expert Security Practices. In *11th Symposium On Usable Privacy and Security (SOUPS 2015)*, pages 327–346, Ottawa, 2015. USENIX Association. 12

[42] P. G. Kelley, L. F. Cranor, and N. Sadeh. Privacy as part of the app decision-making process. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 3393–3402, 2013. 3

[43] J. Kim, Y. Yoon, K. Yi, and J. Shin. ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications. *IEEE Workshop on Mobile Security Technologies (MoST)*, 2012. 3

[44] K. Lee, B. Kaiser, J. Mayer, and A. Narayanan. An Empirical Study of Wireless Carrier Authentication for SIM Swaps. In *16th Symposium on Usable Privacy and Security (SOUPS 2020)*, pages 61–79. USENIX Association, August 2020. 1, 3

[45] K. Lee and A. Narayanan. Security and Privacy Risks of Number Recycling at Mobile Carriers in the United States. In *Symposium on Electronic Crime Research (APWG eCrime)*. IEEE, IEEE, 12/2021 2021. 1, 3

[46] Y. Li, H. Wang, and K. Sun. Email as a master key: Analyzing account recovery in the wild. In *IEEE INFOCOM 2018*, pages 1646–1654. IEEE, 2018. 2

[47] S. G. Lyastani, M. Schilling, M. Neumayr, M. Backes, and S. Bugiel. Is FIDO2 the Kingslayer of User Authentication? A Comparative Usability Study of FIDO2 Passwordless Authentication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 268–285. IEEE, 2020. 2

[48] AbdelKarim Mardini and Guemmy Kim. Making sign-in safer and more convenient. https://web.archive.org/web/20220607220259/https://blog.google/technology/safety-security/making-sign-safer-and-more-convenient/, October 2021. (Accessed on 06/07/2022). 12

[49] W. Melicher, D. Kurilova, S. M. Segreti, P. Kalvani, R. Shay, B. Ur, L. Bauer, N. Christin, L. F. Cranor, and M. L. Mazurek. Usability and Security of Text Passwords on Mobile Devices. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 527–539. ACM. 7

[50] K. Moriarty, B. Kaliski, and A. Rusch. PKCS #5: Password-Based Cryptography Specification Version 2.1. RFC 8018, January 2017. 7

[51] K. Moriarty, M. Nystrom, S. Parkinson, A. Rusch, and M. Scott. PKCS #12: Personal Information Exchange Syntax v1.1. RFC 7292, July 2014. 7

[52] Jakob Nixdorf. andotp deprecation announcement. https://web.archive.org/web/20221004024933/https://forum.xda-developers.com/t/unmaintained-app-4-4-open-source-andotp-open-source-two-factor-authentication-for-android.3636993/page-6#js-post-87021655. (Accessed on 10/03/2022). 13

[53] K. Owens, O. Anise, A. Krauss, and B. Ur. User Perceptions of the Usability and Security of Smartphones as FIDO2 Roaming Authenticators. In *17th Symposium on Usable Privacy and Security (SOUPS 2021)*, pages 57–76, 2021. 2

[54] Can Ozkan and Kemal Bicakci. Security Analysis of Mobile Authenticator Applications. In *2020 International Conference on Information Security and Cryptology (ISCTURKEY)*, pages 18–30. IEEE. 3

[55] C. Percival and S. Josefsson. The scrypt password-based key derivation function. RFC 7914, RFC Editor, August 2016. 7, 13

[56] P. Polleit and M. Spreitzenbarth. Defeating the Secrets of OTP Apps. In *11th International Conference on IT Security Incident Management & IT Forensics (IMF)*, pages 76–88. IEEE, 2018. 3

[57] Ariel Rabkin. Personal Knowledge Questions for Fallback Authentication: Security Questions in the Era of Facebook. In *Proceedings of the 4th Symposium on Usable Privacy and Security*, SOUPS '08, pages 13–23, New York, NY, USA, 2008. Association for Computing Machinery. 2

[58] S. Raponi and R. Di Pietro. A longitudinal study on web-sites password management (in) security: Evidence and remedies. *IEEE Access*, 8:52075–52090, 2020. 2

[59] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill. Studying TLS usage in Android apps. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 350–362, 2017. 3

[60] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman. 50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System. In *Proceedings of the 28th USENIX Security Symposium*, pages 603–620, 2019. 4

[61] E. M. Redmiles and E. Hargittai. New phone, who dis? Modeling millennials' backup behavior. *ACM Transactions on the Web (TWEB)*, 13(1):1–14, 2018. 5

[62] E. M. Redmiles, S. Kross, and M. L. Mazurek. How I Learned to Be Secure: A Census-Representative Survey of Security Advice Sources and Behavior. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 666–677, New York, NY, USA, 2016. ACM. 12

[63] Irwin Reyes, Primal Wijesekera, Joel Reardon, Amit Elazari Bar On, Abbas Razaghpanah, Narseo Vallina-Rodriguez, and Serge Egelman. "Won't Somebody Think of the Children?" Examining COPPA Compliance at Scale. *Proceedings on Privacy Enhancing Technologies*, (2018.3):63–83, 2018. 4

[64] J. Reynolds, T. Smith, K. Reese, L. Dickinson, S. Ruoti, and K. Seamons. A tale of two studies: The best and worst of yubikey usability. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 872–888. IEEE, 2018. 2

[65] Salesforce. Back Up Your Connected Accounts in the Salesforce Authenticator Mobile App. https://help.salesforce.com/s/articleView?id=sf.salesforce_authenticator_backup.htm&type=5. (Accessed on 10/03/2022). 8, 14

[66] Salesforce. Restore Connected Accounts in the Salesforce Authenticator Mobile App. https://help.salesforce.com/s/articleView?id=sf.salesforce_authenticator_restore_from_backup.htm&type=5. (Accessed on 10/03/2022). 14

[67] Stuart Schechter, A.J. Bernheim Brush, and Serge Egelman. It's No Secret. Measuring the Security and Reliability of Authentication via "Secret" Questions. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, pages 375–390, Los Alamitos, CA, USA, 2009. IEEE Computer Society. 2

[68] 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*, page 2. ACM, 2010. 1

[69] Keira Stevens. Hashes, salts, and rainbow tables: Confessions of a password cracker. Dark Reading, May 12 2021. https://www.darkreading.com/application-security/hashes-salts-and-rainbow-tables-confessions-of-a-password-cracker/a/d-id/1340928. 7

[70] J. Tan, K. Nguyen, M. Theodorides, H. Negron-Arroyo, C. Thompson, S. Egelman, and D. Wagner. The Effect of Developer-Specified Explanations for Permission Requests on Smartphone User Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2014. 3

[71] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki, et al. Data breaches, phishing, or malware?: Understanding the risks of stolen credentials. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 1421–1434. ACM, 2017. 1

[72] C. Thompson, M. Johnson, S. Egelman, D. Wagner, and J. King. When It's Better to Ask Forgiveness than Get Permission: Designing Usable Audit Mechanisms for Mobile Permissions. In *Proceedings of the 2013 Symposium on Usable Privacy and Security (SOUPS)*, 2013. 3

[73] L. Tsai, P. Wijesekera, J. Reardon, I. Reyes, S. Egelman, D. Wagner, N. Good, and J. Chen. Turtle Guard: Helping Android Users Apply Contextual Privacy Preferences. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 145–162, Santa Clara, CA, 2017. USENIX Association. 3

[74] B. Ur, F. Alfieri, M. Aung, L. Bauer, N. Christin, J. Colnago, L. F. Cranor, H. Dixon, P. Emami Naeini, H. Habib, N. Johnson, and W. Melicher. Design and Evaluation of a Data-Driven Password Meter. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 3775–3786. ACM. 7

[75] Emanuel von Zezschwitz, Alexander De Luca, and Heinrich Hussmann. Honey, I shrunk the keys: Influences of mobile devices on password composition and authentication performance. In *Proceedings of the 8th Nordic Conference on Human-Computer Interaction: Fun, Fast, Foundational*, pages 461–470. ACM. 7

[76] Alex Weinert. How it works: Backup and restore for microsoft authenticator. Available: https://web.archive.org/web/20220630215634/https://techcommunity.microsoft.com/t5/microsoft-entra-azure-ad-blog/how-it-works-backup-and-restore-for-microsoft-authenticator/ba-p/1006678. [Online; accessed: 30-Jun-2022]. 8

[77] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android Permissions Remystified: A Field Study on Contextual Integrity. In *Proc. of USENIX Security*, 2015. 3, 4

[78] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov. The Feasability of Dynamically Granted Permissions: Aligning Mobile Privacy with User Preferences. In *Proc. of IEEE Symposium on Security and Privacy (SP)*, Oakland '17, 2017. 3, 4

[79] P. Wijesekera, J. Reardon, I. Reyes, L. Tsai, J. Chen, N. Good, D. Wagner, K. Beznosov, and S. Egelman. Contextualizing Privacy Decisions for Better Prediction (and Protection). In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 1–13, New York, NY, USA, 2018. Association for Computing Machinery. 3

[80] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. Sadeh, S.M. Bellovin, and J.R. Reidenberg. Automated Analysis of Privacy Requirements for Mobile Apps. In *Proc. of NDSS Symposium*, 2017. 3

[81] Zoho. Secure non-Zoho accounts using OneAuth's OTP authenticator. https://web.archive.org/web/20221003160327/https://help.zoho.com/portal/en/kb/accounts/oneauth/v2/articles/otp-authenticator#Back_up_and_restore_OTP_secrets. (Accessed on 10/03/2022). 8

A Base32 decryption heuristic

Given the ciphertext of the encrypted TOTP secret, what is the probability that a single password guess will generate a plaintext output that is valid Base32? An ASCII character has $2^8 = 256$ possible bit permutations and Base32 allows 32 valid characters (A-Z and 2-7). The probability that an L-length ASCII string is valid Base32 is:

$$= P(\text{single byte is valid Base}32)^L = (32/256)^L = 0.125^L$$

The probability that a single password guess for a 32 byte TOTP secret (L = 32), which is a common length used in industry, will generate valid Base32 is: $= 0.125^{32} \approx 1.26 * 10^{-29}$. With a very high probability, this heuristic will accurately verify whether the user entered the correct recovery password because it is extremely unlikely that the decryption process will result in plaintext that is valid Base32 format if the encryption key is derived from an incorrect password.