Easy Encryption for Email, Photo, and Other Cloud Services

John Seunghyun Koh

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2021

Abstract

Easy Encryption for Email, Photo, and Other Cloud Services

John Seunghyun Koh

Modern users carry mobile devices with them at nearly all times, and this likely has contributed to the rapid growth of private user data—such as emails, photos, and more—stored online in the cloud. Unfortunately, the security of many cloud services for user data is lacking, and the vast amount of user data stored in the cloud is an attractive target for adversaries. Even a single compromise of a user's account yields all its data to attackers. A breach of an unencrypted email account gives the attacker full access to years, even decades, of emails. Ideally, users would encrypt their data to prevent this. However, encrypting data at rest has long been considered too difficult for users, even technical ones, mainly due to the confusing nature of managing cryptographic keys.

My thesis is that strong security can be made easy to use through client-side encryption using self-generated per-device cryptographic keys, such that user data in cloud services is well protected, encryption is transparent and largely unnoticeable to users even on multiple devices, and encryption can be used with existing services without any server-side modifications. This dissertation introduces a new paradigm for usable cryptographic key management, Per-Device Keys (PDK), and explores how self-generated keys unique to every device can enable new client-side encryption schemes that are compatible with existing online services yet are transparent to users. PDK's design based on self-generated keys allows them to stay on each device and never leave them. Management of these self-generated keys can be shown to users as a device management abstraction which looks like pairing devices with each other, and not any form of cryptographic key management. I design, implement, and evaluate three client-side encryption schemes supported by PDK, with a focus on designing around usability to bring transparent encryption to users.

First, I introduce Easy Email Encryption (E3), a secure email solution that is easy to use. Users struggle with using end-to-end encrypted email, such as PGP and S/MIME, because it requires users to understand cryptographic key exchanges to send encrypted emails. E3 eliminates this key exchange by focusing on storing encrypting emails instead of sending them. E3 transparently

encrypts emails on receipt, ensuring that all emails received before a compromise are protected from attack, and relies on widely-used TLS connections to protect in-flight emails. Emails are encrypted using self-generated keys, which are completely hidden from the user and do not need to be exchanged with other users, alleviating the burden of users having to know how to use and manage them. E3 encrypts on the client, making it easy to deploy because it requires no server or protocol changes and is compatible with any existing email service. Experimental results show that E3 is compatible with existing IMAP email services, including Gmail and Yahoo!, and has good performance for common email operations. Results of a user study show that E3 provides much stronger security guarantees than current practice yet is much easier to use than end-to-end encrypted email such as PGP.

Second, I introduce Easy Secure Photos (ESP), an easy-to-use system that enables photos to be encrypted and stored using existing cloud photo services. Users cannot store encrypted photos in services like Google Photos because these services only allow users to upload valid images such as JPEG images, but typical encryption methods do not retain image file formats for the encrypted versions and are not compatible with image processing such as image compression. ESP introduces a new image encryption technique that outputs valid encrypted JPEG files which are accepted by cloud photo services, and are robust against compression. The photos are encrypted using self-generated keys before being uploaded to cloud photo services, and are decrypted when downloaded to users' devices. Similar to E3, ESP hides all the details of encryption/decryption and key management from the user. Since all crypto operations happen in the user's photo app, ESP requires no changes to existing cloud photo services, making it easy to deploy. Experimental results and user studies show that ESP encryption is robust against attack techniques, exhibits acceptable performance overheads, and is simple for users to set up and use.

Third, I introduce Easy Device-based Passwords (EDP), a password manager with improved security guarantees over existing ones while maintaining their familiar usage models. To encrypt and decrypt user passwords, existing password managers rely on weak, human-generated master passwords which are easy to use but easily broken. EDP introduces a new approach using

self-generated keys to encrypt passwords, and an easy-to-use pairing mechanism to allow users to access passwords across multiple devices. Keys are not exposed to users and users do not need to know anything about key management. EDP is the first password manager that secures passwords even with untrusted servers, protecting against server break-ins and password database leaks. Experimental results and a user study show that EDP ensures password security with untrusted servers and infrastructure, has comparable performance to existing password managers, and is considered usable by users.

Table of Contents

Acknowledgements

I give special thanks to my advisors, Steven M. Bellovin and Jason Nieh. Thank you for all your support over the years before, during, and after the doctoral program. I could not have done it without your input and constructive criticism on my work, and also your support for me as colleagues and fellow human beings. I also greatly appreciate the other members of my committee—Roxana Geambasu, Asaf Cidon, and Moti Yung—for taking the time and effort to review my dissertation and thesis defense.

I'd also like to thank my fellow doctoral students, in no particular order: Naser AlDuaij, Shih-Wei Li, Jintack Lim, Alex Van't Hof, Vaggelis Atlidakis, Mathias Lécuyer, and many more. Without all your advice, feedback, and even small talk, it would have been a very lonely five years. My research was supported in part by NSF grants CNS-1717801 and CNS-1563555, and a Google Faculty Research Award.

I would also like to thank my mother and father for all of their love and support, and for believing in me.

To my mother and father

Chapter 1: Introduction

The rapid proliferation of always on and always connected mobile devices has driven an explosive growth in the amount of information that users produce, store, and communicate or share online. This has been further accelerated by the growing number of capabilities and features supported by modern smartphones and tablets that allow users to produce daily streams of personalized content. In response, online services have risen to the task of providing affordable yet highly available, reliable, and persistent online infrastructure—often referred to as the cloud—to support the communication and storage of user data. Users increasingly trust these reliable cloud-based online services to safeguard, and in many cases, permanently store their private and sensitive data, such as emails, photos, videos, documents, and much more. This is problematic for users because they store years and sometimes even decades worth of personal data in the cloud, and all it takes is a single successful compromise of their accounts to reveal all of their data.

The issue is that the security of online services is poor due to the lack of encryption. They often only rely on easily compromised user account credentials—usernames and passwords—to authenticate users, and it is a simple matter to find massive databases containing working mappings of usernames and passwords, often procured via password database leaks and hacks. Worse yet, users often reuse these credentials for multiple services, thus amplifying the impacts of such leaks [1, 2]. Even without these lists, most passwords are easy to guess despite key-strengthening techniques since after all, they are generated by humans [3, 4, 5]. Some services augment the user credentials check with two-factor authentication (2FA) which most often requires the user to provide proof of ownership of a trusted secret ("something you own"), but this only defends against account-based compromises. The reality is that adversaries can entirely bypass any account-based authentication checks, such as by compromising the servers of an online service or by simply being the service itself. Some examples of server-side compromises include but are not limited to:

- an external hacker exploiting a bug in a server to gain access to the data stored on it,
- another external attacker using phishing or social engineering to obtain an employee's credentials at the service provider,
- a law enforcement agency seizing servers as evidence for a case,
- an internal adversary such as a rogue employee abusing privileged access to user data [6, 7, 8, 9],
- or even the service itself by design, such as "honest-but-curious" providers who have legitimate access to private user data and take the opportunity to aggressively mine and analyze it [10, 11].

In all these cases, user account credentials are irrelevant, the privacy of user data is compromised, and users are powerless to defend themselves.

A way which users could protect themselves is through the use of cryptography [12]. If users encrypt their private data with a secret key known to only them, this would greatly reduce the possible attack surface area, addressing all of the attack vectors listed prior. The problem is that the vast majority of popular online services do not offer such an option. Although there are various reasons for this, one of the primary factors is that the average user has no idea how to use cryptography. It has been traditionally difficult and confusing to use, as shown time and time again for over two decades in a variety of contexts and applications [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24], including email, instant messaging, social networking services, and even tactical radio traffic used by trained US federal law enforcement officers. In recent years, Apple's FileVault [25] for macOS and Microsoft's BitLocker [26] have set the bar for providing disk encryption to average end users, and have made significant progress in easing the difficulty of using it. The way they achieved this was by greatly simplifying the usage model: users only need to perform a simple step of enabling the disk encryption and perhaps writing down a recovery key or linking the configuration to their cloud service account with Apple for FileVault, or with Microsoft for BitLocker. Thereafter,

all encryption and decryption happens transparently to the user with no further interaction. This simple setup and usage model is possible primarily due to one important design decision: each FileVault or BitLocker encryption key is tied to a single local device, without any encryption of remote data. In other words, when it comes to the average end user, FileVault and BitLocker do not use their keys for multiple device encryption and decryption because it is unnecessary, so their keys never need to be moved to other devices. This makes their usage models simple to use and easy to understand.

However, in most cases of applied cryptography beyond local disk encryption, support for multiple device encryption and decryption of both locally or remotely stored data is an important requirement. In turn, supporting users with multiple devices requires a solution for cryptographic key management because in the traditional cryptography model, each device needs access to the same encryption key or set of keys; this allows each device to independently encrypt and decrypt the user's data. Cryptographic key management is also what users struggle with the most. Users do not understand it, and encounter severe usability issues related to selecting the correct keys and actually encrypting data; users essentially lack the requisite knowledge for understanding how to store, transfer, and use cryptographic keys [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]. Consider traditional encrypted email such as PGP. A PGP email sender encrypts an email using the PGP public key of an intended recipient, and the recipient may check their email on any one of their multiple devices which needs to decrypt the email with the PGP private key. This already places two highly technical burdens on the involved users. First, the sender must somehow obtain the correct public key of the recipient which is necessary to encrypt the sent email, and the recipient must have copied their private key to all their devices to be able to decrypt the email on any of them. This cryptographic exchange significantly differs from the much simpler case of FileVault and BitLocker because users must understand managing crytographic keys not only locally on their devices, but also when securely exchanging public keys with other users.

Any service that supports encryption for users also needs to support cryptographic key management features, but few, if any, do. The issues are a lack of usable yet platform-independent

cryptographic key management solutions, and expecting users to know how to utilize encryption at all; if these problems are addressed, then encryption should be much more accessible to users. But providing usable cryptographic key management with comprehensible encryption models has proven to be a difficult problem to solve, so most services have settled for only requiring username and passwords for security despite a long history of proposals to replace them [27]. This stubborn reliance on only passwords and 2FA is insufficient since the threat is no longer just account-based compromises, but server-side ones and even the services themselves. The current account-based security paradigm for data security for users may have been suitable many years ago, but is no longer so as evidenced by the billions of breached and leaked passwords on public databases [28].

My thesis is that strong security can be made easy to use through client-side encryption using self-generated per-device cryptographic keys, such that user data in cloud services is well protected, encryption is transparent and largely unnoticeable to users even on multiple devices, and encryption can be used with existing services without any server-side modifications. This is contrary to the widely held belief that strong encryption is unusable by average users due to requiring an in-depth understanding of how encryption and its keys operate. I designed, implemented, and evaluated a novel cryptographic key management approach based on self-generated keys call Per-Device Keys (PDK) for improved data security that sacrifices little to no usability. Self-generated PDK keys stay on each device and users never need to know about them. For a given user, any device can be used to encrypt the user's data such that any other device belonging to the user can also decrypt the data. The PDK design using self-generated keys achieves the simple usage model of systems like FileVault and BitLocker, but unlike these systems, supports encryption and decryption on multiple devices for even remote data, not just local data. My use of self-generated keys also shows that encrypting data does not necessarily require trust provided by a third party, in contrast to competing solutions which rely on public key infrastructure and services as seen in Apple's approach based on having Apple devices trust the iCloud secure architecture. These aspects of PDK are made possible in recent years by leveraging developments in mobile computing usage patterns by the average user; in essence, users carry mobile devices with them at all times which

are trusted and always connected to the Internet, and which are now powerful enough to encrypt data in a timely fashion. In other words, my thesis, which could not be realized two decades ago, is now made possible due to technological advancements and cultural shifts in mobile computing.

I combine PDK with client-side encryption of user data using public key cryptography with existing services. At a high level, this consists of having all of a user's devices generate private/public keypairs, locally encrypting their private data using all of the public keys belonging to the user's devices, and uploading the encrypted data to online services while still providing a seamless experience to users that exhibits little difference from a regular client with no encryption. It is this concept of generating a new private/public keypair on each of a user's devices, and also the associated key management system which defines PDK.

My general system design is applicable to a broad spectrum of online services where users store their private and sensitive data. The system design guarantees the confidentiality and integrity of users' private data with minimal overhead with regards to both performance and usability, under various threat model and trust assumptions including both trusted and untrusted servers and infrastructure, with no reliance on custom secure hardware. A key insight and motivator for this dissertation is addressing a long-standing and problematic assumption made by existing applications that use encryption: the assumption that if users are using an application with encryption, then they probably are security-conscious and therefore already understand cryptography and secure infrastructure. A secondary problematic assumption for secure applications is that online services cooperate with and are compatible with the encryption being used, which is not necessarily the case in practice. In contrast, my proposed design makes no such assumptions about users or the cooperation of services.

By addressing the usability issues with cryptographic key management, I am then able to explore the possibilities for client-side encryption of user data, namely in the context of encrypting user data in a way that is compatible with existing cloud services. Such a solution provides numerous benefits, including no reliance on and complete privacy from any third party services or devices to perform the encryption on behalf of users, and quick deployment with a low barrier to entry as

users only need to install a client app and begin encrypting their data immediately. This is beneficial to both developers and users because no work needs to be done to existing servers, back ends, and protocols, and because users experience only minor changes in their typical usage patterns when compared to a regular app with no encryption. The key issue to address is designing encryption solutions within these constraints but also the limtiations of the cloud services themselves. For example, photo services such as Google Photos only accept files which are valid images, whereas encrypted data often appears to be entirely random bits, necessitating an encryption algorithm that outputs ciphertexts which are valid images. Furthermore, an important requirement is that all the encryption and decryption is transparent to the user and happens automatically behind the scenes so that the user is not burdened with cryptographic concepts and potential significant performance overhead.

I introduce three systems in which I design, implement, and evaluate the PDK design when coupled with transparent and strong client-side encryption. The three systems exhibit the effectiveness of my system in varying contexts and environments which consist of three popular online services: one for email, one for photos, and one for passwords. They are representative of different threat and trust models for infrastructure with differing encryption requirements with respect to their formats and keying material.

First, I present Easy Email Encryption (E3) [29], an easy-to-use encrypted email app. Users traditionally struggle with end-to-end encrypted email solutions such as PGP and S/MIME due to their heavy reliance on users understanding complex concepts in cryptography such as cryptographic key exchanges. E3 has no such requirement for users because it uses self-generated keys which are entirely internal to its system in a way that completely hides them from users. This design choice allows E3 to eliminate confusing cryptographic key exchanges. These self-generated keys are then used to only encrypt stored emails when they are received instead of when they are sent, which protects them against email account and server compromises. In-flight emails being sent over the network are protected using widely-used securely encrypted TLS connections. E3 improves the usability of secure email because it only exposes a device pairing abstraction to

users. This design is applicable to a threat and trust model where users initially trust the servers and infrastructure, which may become malicious or compromised at a later time. E3 is compatible with any standard IMAP server including all of the most popular email providers like Gmail and Yahoo!, leverages existing encryption formats for email—PGP and S/MIME—for the client-side encryption, and introduces the first concrete implementation of the PDK key management concept. User study results for E3 show that E3 provides improved security guarantees over the norm of unencrypted email yet is much easier to use than end-to-end encrypted email such as PGP.

Second, I present Easy Secure Photos (ESP) [30], an encrypted photo app that allows users to encrypt their photos and store them in existing cloud photo services. Users normally cannot encrypt their photos for storage in widely used cloud services such as Google Photos because typical encryption methods do not retain valid image file formats, or if they do, they are not compatible with compression of alterations of the iamge ciphertexts. ESP uses a new image format-preserving encryption technique to produce encrypted JPEG images that are compatible with existing photo hosting services, with no changes to servers or protocols. ESP users only need to install a ESP app, making it easy to deploy. Like E3, uses self-generated keys to encrypt and decrypt images in a similar manner as E3 does for email, such that all cryptographic key management is hidden from the user. Also like E3, ESP's threat and trust model assume that the servers and infrastructure, such as the servers hosting users' images, are initially trusted. However, there are some key differences between the two: (1) The use of a new format-preserving encryption method provides a case study for the motivations and design decisions required to support client-side encryption for online services which restrict the file type formats of user-uploaded data. (2) Since ESP uses its own format-preserving encryption scheme, it uses a different security model from E3 and applies the concept of using PDK keys as key-encrypting keys. (3) ESP's design for PDK utilizes an unconventional format for its key management protocol messages, showing that it is flexible enough for not only online services based on textual communication, but can be used in a variety of contexts and communications channels with a little creativity. The experimental results and user studies show that ESP is secure, has acceptable performance overheads, and is simple for users to

use.

Third, I present Easy Device-based Passwords (EDP), a password manager system and client with improved security guarantees over existing ones while maintaining their familiar usage models. Existing password mangers rely on weak, human-generated master passwords to encrypt user passwords because this usage model is easy for users to understand. The drawback is that master passwords are by definition easy for adversaries to guess and break. EDP introduces a new approach to password management to address the weakness of master passwords by instead using self-generated keys to encrypt passwords. This is then coupled with an easy-to-use device pairing mechanism so that users can enroll EDP devices which gives them access to their passwords on their devices. Like with E3 and ESP, the self-generated keys are internal to the EDP system and are not exposed to users, so users do not need to understand any cryptographic key management concepts. EDP is the first password manager that secures passwords even when using untrusted servers for storing passwords, making EDP also the first password manager that protects against server compromises and password database leaks as an inherent part of its design. This is in contrast to existing password managers which take extra precautions to protect users' weakly encrypted passwords which are vulnerable to break-ins and leaks. EDP also operates securely with the use of untrusted connections, and is able to manage users' PDK keys even when their devices communicate over malicious communications channels. The experimental results and user study for EDP show that it ensures password security under these conditions, has comparable performance to existing password managers, and is considered usable by users.

The contributions of this dissertation include:

1. The design, implementation, and evaluation of a novel cryptographic key management scheme, Per-Device Keys (PDK), which puts a new spin on how to approach the long outstanding issue of usable key management by leveraging self-generated keys and modern advancements in mobile computing.

2. A blueprint for using PDK to support new and usable client-side encryption models with existing widely used cloud services even with varying constraints and restrictions imposed

by the services and users' devices.

3. The design, implementation, and evaluation of Easy Email Encryption (E3) which introduces the encrypt-on-receipt approach to secure email for IMAP-based mail services which focuses on encrypting emails when they are stored instead of when they are sent, unlike in end-to-end encrypted email like PGP and S/MIME. This requires no changes to the IMAP protocol or email servers, and adds acceptable performance overhead.

4. How to implement and use PDK in the context of E3, with a device enrollment approach based on a two-way verification step, together with an evaluation of its usability via user studies.

5. The design, implementation, and evaluation of Easy Secure Photos (ESP) which introduces an image encryption algorithm based on using grayscale JPEG images to output valid JPEG images that are robust against ciphertext image compression with imperceptible changes in quality.

6. How to implement and use PDK in the context of ESP, with its cryptographic key management solution which has no reliance on any third party service and uses only the photo service itself.

7. The design, implementation, and evaluation of Easy Device-based Passwords (EDP) which introduces a password manager that improves the security guarantees of typical password managers by encrypting user passwords with PDK keys instead of weak, human-generated master passwords. EDP achieves this while still maintaining the familiar password-based usage model of existing password managers.

8. A significant improvement to the security guarantees of PDK as designed and implemented in EDP which utilizes a Password Authenticated Key Exchange (PAKE) and an untrusted relay server for PDK device verification.

Chapter 2: System Design - Client-side Encryption and Key Management

We present a general system design to address the weak security of online services by giving users agency to control the security of their data. We do this with a new usable key management design that appears only as a device management scheme to users, which then provides the foundation to enable them to use strong client-side encryption. We show that this system is flexible enough to meet the needs of nearly any online service that stores users' private data without adding significant overhead to performance and usability metrics.

We define private data as any piece of data for which a user wishes to preserve confidentiality and integrity. Some examples of private data include user emails, photos, and passwords. Traditionally, online services have stored private data on their servers with limited security; most services only protect user data through basic password-based authentication and authorization requirements, i.e. requiring users to provide an authorized username and password to authenticate themselves and gain access to their data. Some services apply server-side encryption of data at rest on their disks, but this is opaque to users and attackers in the sense that the simple username and password combination is enough to bypass server-side encryption. Moreover, server-side encrypted data is fully accessible to the online service itself which may be or become malicious. These issues motivate the use of strong, client-side encryption which addresses the use of weak passwords by users, and also protects user private data against the prying eyes of adversaries.

2.1 Key Management via Per-Device Keys

One of the goals of this dissertation is to enable the use of strong encryption, such as public key cryptography, for users. Public key cryptography consists of private/public keys which are used to encrypt/decrypt data. Using public keys to encrypt data sounds straightforward, but has

famously posed a difficult problem in terms of usability for non-technical, average users who struggle with understanding public key cryptography and how to use it [13]. Typically, users are tasked with managing their keypairs which entails numerous activities such as storing, retrieving, authenticating, and correctly using them. These management requirements are often far beyond the capabilities of the average user; normally, some of the requirements such as authentication are addressed using public key infrastructure (PKI), but PKI is even more confusing to users as it introduces many complicated concepts and jargon that often mystify even technically literate users.

Another difficulty of using public key cryptography (and cryptography in general) is the question of how private keys (or secret keys) are transferred and synchronized across a user's multiple devices. The usual usage model for private/public keypairs is that a user generates a single one which is then copied to all of their devices which need the private key for decryption purposes. However, there are many pitfalls associated with moving or copying private keys around due to the risk of a user accidentally revealing the keys during the process, and these dangers are even more prominent for the average user who lacks a conceptual understanding of the security issues. Essentially, key management is difficult for not only average users but even technical ones as well. The PDK concept and design, which addresses these issues, is one of the main contributions of this dissertation, and the implementations and evaluations of it in real applications show that it greatly simplifies public key cryptography's key management for even average users. Furthermore, PDK gives users agency in their security, as all the cryptographic operations are in the hands of the user, and not service providers or custom hardware.

The design for PDK consists of generating a keypair on each of a user's multiple devices, so that each device has its own unique one. PDK's primary features are as follows:

(1) Private keys never move or leave a device.

(2) A private key is "revoked" by re-encrypting data to all public keys except for the revoked one.

(3) Private key recovery, normally mitigated with private key backups, is reduced to device data backup which is easier. As long as one device is available, encrypted user data can be decrypted without a recovery process.

(4) Public keys are automatically distributed using an authenticated communications channel.

(5) Keys are self-generated (and self-signed in the case of certificates) so users can freely add new devices.

(6) Private keys use local secure storage when available without relying on a user password so there is no password to target in phishing attacks.

These features allow us to avoid one of the biggest issues usually associated with public key cryptography which is the question of how to manage and synchronize the private key across a user's multiple devices. Since each device has its own private/public keypair, users never need to transfer or copy the private key to other devices. This approach reduces the key management problem from having to deal with private keys, to having to synchronize public keys, which has fewer possible security pitfalls since public keys are safe to be revealed. This is in effect a return to traditional security best practices which advise users to never transport private keys because doing so is insecure. This advice is almost never followed in practice because users often access their encrypted data from multiple devices, all of which need the same private key when using common encryption usage models. But in PDK, when a user's client encrypts their data, it does so using all of the public keys from each of the user's devices. Therefore, the first and foremost design requirement for PDK is having a method for users to synchronize their devices' public keys.

Although synchronizing public keys requires fewer security measures compared to private keys, the synchronization process still needs to satisfy an important security property which is that the public keys are authenticated. Any public keys that are used to encrypt user data must be verified to belong to the user so that they can be considered trusted. Otherwise, an attacker could provide a malicious public key and trick users into encrypting their data with it, which would give the attacker the capability to decrypt the users' data. We thus wish to create a fully connected

Key Management	Device Management	Description
Generate keypair	Configure new device	When a user installs an app that uses PDK, it informs the user that it is configuring the device. This entails generating a new PDK keypair.
Authenticate public key	Add/enroll new device	When a user adds a new PDK device, the existing device authenticates the public key of the new device being added.
Exchange public keys	Add/enroll new device	Also a part of adding a new device. After a new device's public key is authenticated, the devices can exchange their known trusted public keys.
Revoke keypair	Delete device	When a user deletes a device, the device's public key is no longer trusted and this update is propagated to all trusted devices.
Re-key device	Re-configure device	A device may need a new keypair if a crypto algorithm has become insecure or the key was somehow compromised. This may also be shown to users as an "update device security" concept or variations of it.

Table 2.1: Common key management functions converted to a device management abstraction.

graph of trust among all of a user's devices such that each device trusts every other device's public key belonging to the user, but no others. To do this, users authenticate individual PDK public keys when they add or enroll a new device with their PDK ecosystem.

The exact design for a user to authenticate PDK public keys depends on the context of a given solution; the approach will change depending on cloud service and data being encrypted. However, the general design should satisfy the following criteria:

(1) All key management functions should be presented to users in the context of device management, as summarized in Table 2.1. For example, the act of authenticating a PDK public key should appear to the user as enrolling a new device, i.e. users should not be exposed to the concept of keys. Users enroll a new device using an existing device; if there is no existing device because the new device is the user's first one, then this first device becomes automatically trusted.

(2) There should be a secure communication channel among a user's devices, which ideally does not require providing log in credentials to a third party service. PDK devices use this channel

to authenticate and exchange newly and previously authenticated public keys.

(3) There should be a recovery mechanism for when users' devices are lost or compromised and need to be revoked.

The solutions for selecting and constructing a secure communication channel, and the process for authenticating new device PDK public keys are the primary differentiators of PDK designs, while the requirement of mapping key management concepts to device management ones is a more generalizable decision concerning user experience and interface. Users naturally understand the device management abstraction of synchronizing their devices, and using them to encrypt and decrypt their data. It is simple and intuitive, and perhaps most importantly, a commonly used abstraction among consumer devices. For example, Apple has its users synchronize their new devices via iCloud, which involves many high security systems and complicated architectures behind the scenes, but users are only exposed to the device management view. In contrast, traditional cryptographic key management requires users to understand the technical security jargon and concepts underlying the keys, but the issue is average users do not comprehend the differences between private and public keys, how to keep the private keys secure while copying them to their multiple devices, and how to securely exchange public keys with other users. For users to use a secure system that uses cryptographic keys, it is imperative for the experience to be as easy to understand and use as possible, and my system designs which use PDK achieve this via exposing a device management abstraction focused on a smooth user experience.

2.2 Client-side Encryption

The challenge of client-side encryption is designing it in such a way that any encrypted data is compatible with services that are not designed for it, and so that it is transparent to users in that they do not need to know the finer details of cryptographic operations. Addressing the first challenge of compatability with existing services overcomes a common drawback of many proposed encryption systems which is their reliance on server-side changes and support for encryption. This drawback

may also manifest as a system which relies on a third party service to which to offload encryption and decryption. The second challenge of making encryption transparent to users is closely tied to the use of PDK for cryptographic key management. If the keys can be seamlessly managed, then it is possible to automatically encrypt and decrypt user data on multiple devices without manual configuration beyond simple device management.

The requirement for strong encryption is satisfied by using public key or asymmetric cryptography with the PDK private/public keypairs belonging to and maintained by users. Since PDK keys are wholly managed by users, their clients manage all encryption and decryption operations. Some online services may be compatible with existing public key-based ciphertext formats, such as email which has standardized encrypted email formats, but others may have file format restrictions which necessitate the use of unique format-preserving encryption strategies such as for encrypted images stored on photo hosting services. Regardless of the exact ciphertext format, the public keys are used by users' clients to encrypt their data either in a standard hybrid cryptosystem scheme, or in a key-encrypting key scheme to encrypt any secret keys or values necessary to encrypt data that have special format requirements.

The standard hybrid cryptosystem scheme consists of using public keys to encrypt cryptographically secure, randomly generated AES keys which in turn are used to encrypt data using symmetric encryption. The motivation for the hybrid design is mainly performance, as public key cryptographic operations are inefficient in comparison to ones based on symmetric key cryptography such as AES. Technically, the hybrid cryptosystem is a key-encrypting key scheme, but is most commonly associated with encrypting specifically AES keys. Instead, the public keys may be used to encrypt other kinds of secret values or keys, not just AES keys. Thus, the client-side encryption may also opt to use the public keys as key-encrypting keys in a more general sense.

Chapter 3: Easy Email Encryption (E3)

3.1 Introduction

Email accounts and servers are an attractive target for adversaries. They contain troves of valuable private information dating to years back, yet are easy to compromise. Some prominent examples include: the phishing attack on Hillary Clinton's top campaign advisor John Podesta [31], the 2016 email hack of one of Vladimir Putin's top aides [32], the email leaks of former Vice President candidate Sarah Palin and CIA Director John Brennan [33, 34], and other similar cases [35]. These attacks targeted high profile individuals and organizations to leak their emails and damage their reputations. In the Podesta leaks, attackers perpetrated a spear-phishing attack to obtain John Podesta's Gmail login credentials, access his emails, and leak them to WikiLeaks. Sarah Palin was subjected to a simple password recovery and reset attack which granted the attacker full access to her personal email account on the Yahoo! Mail website. John Brennan's AOL web email account was compromised via social engineering. Adversaries also sometimes seize entire email servers such as in the cases of cock.li and TorMail [36, 37], or compromise them, such as in the Sony Pictures email leaks [38].

The common thread is that a compromise exposes the *entire history* of affected users' emails after a single breach. With the explosive growth in cloud storage, it is easy to keep gigabytes of old emails at no cost. Gmail's massive storage capacity—up to 15 GB for free, or 30 TB for paid options [39]—opens up the possibility of keeping email forever. Consequently, users often email themselves to use their inbox as backup storage for important information, thereby exacerbating the cost of a compromise.

Existing secure email models based on end-to-end encryption are thought to be effective against attackers but are rarely used. Examples include Pretty Good Privacy (PGP) [40], and Secure/Mul-

tipurpose Internet Mail Extensions (S/MIME). Both are too complicated for most users because all email correspondents must comprehend public key cryptography. The current paradigm places too much of a burden on senders who must correctly encrypt emails and manage keys [18, 13]. The result is even technical users rarely encrypt their email. End-to-end encryption for email seeks absolute security[1] at the expense of usability, creating a chasm between absolute security via encrypting all emails via PGP or S/MIME, and protecting no emails at all.

We introduce an approach to encrypted email that addresses the gaping void between unusable but absolute security, and usable but no security. We change the problem from sending encrypted emails to *storing* them since it is a user's *history* of emails that is most tantalizing to attackers. Our goal is to mitigate the attacks often publicized in the news where email account credentials or servers are compromised. The attackers have access to emails stored on servers but not individual devices. Most of the attacks are either simplistic phishing attacks for email account credentials or server breaches that include innocent users in the collateral damage. All the affected emails would be protected had they been encrypted prior to any breach using a key inaccessible to the email service provider. We therefore seek a client-side encrypted email solution that safeguards any emails received prior to a compromise. Furthermore, such a defense must be usable for non-technical users, and compatible with correspondents who do not use encrypted email.

We present Easy Email Encryption (E3) as the first step to filling this void. E3 provides a client-side encrypt-on-receipt mechanism that makes it easy for users as they do not need to rely on public key infrastructure (PKI) or coordinate with recipients. The onus is no longer on the sender to figure out how to use PGP or S/MIME. Instead, email clients automatically encrypt received email without user intervention. E3 protects all emails received prior to any email account or server compromise for the emails' lifetime, with threat models similar to those of more complex schemes such as PGP and S/MIME; for ease of discussion we hereafter refer to PGP and S/MIME email as end-to-end encrypted email.

E3 is designed to be compatible with existing IMAP servers and IMAP clients to ease the

[1] Absolute security refers to the strength of the cryptographic primitives, but not mail client implementations which contain vulnerabilities and bugs that can compromise end-to-end encrypted email [41].

adoption process. An E3 client downloads messages from an IMAP server, encrypts them in a standard format, and uploads the encrypted versions. The original cleartext emails are then deleted from the server. No changes to any IMAP servers are necessary. Users require only a single E3 client program to perform the encryption. Existing mail clients do not need to be modified and can be used as-is alongside a separate E3 background app or add-on. If desired, existing mail clients can be retrofitted with E3 instead of relying on a separate app or on an add-on.

Users are free to use their existing, unmodified mail clients to read E3-encrypted email if they support standard encrypted email formats. The vast majority of email clients support encrypted emails either natively or via add-ons. Other than the added security benefits of encryption, all functionality looks and feels the same as a typical email client, including spam filtering and having robust client-side search capability.

Key management, including key recovery, is simplified by *Per-Device Keys (PDK)* management which provides significant benefits for the common email use case of having two or more devices for accessing email, e.g. desktop and mobile device mail clients. Users with multiple devices leverage PDK with no reliance on external services. Users who truly only use a single device still benefit from PDK's key configuration and management capabilities, but rely on free and reliable cloud storage for recovery. E3 as a whole is a usable solution for encrypted email that protects a user's history of emails while also providing a simple platform-independent key management scheme.

E3 is easy to implement and use. We have implemented it for multiple environments, including retrofitting existing Android mail clients with E3 for use with mobile devices, implementing an extension for the Google Chrome web browser to use E3 as a Gmail web client, and implementing a daemon-like Python client that allows users to use existing unmodified mail clients. We tested that the Android and Python prototypes work with popular email services, including Gmail, Yahoo! Mail, and AOL Mail. We also quantified the performance of E3 on Android. Our measurements show that while E3 imposes a one-time cost for email encryption, the total overhead is quite reasonable from a user perspective. Finally, we present the results of a user study for E3 that show

that users consider it simple, intuitive, and flexible.

3.2 Threat Model

The purpose of E3 is to protect all emails stored *prior* to any email account or server compromise, with no software or protocol changes except for installing E3 itself on a recipient's devices. The primary risk we defend against is to stored mail on the IMAP server. If the account or server is compromised, all unencrypted mail is available to the attacker.

We thus guard against *future* compromise of the user's IMAP account or server. We assume that the IMAP account and server are initially secure, and that at some later time, one or both are compromised. We therefore assume that email services are honest; the threat is external entities trying to access email account data. If email service providers are not honest, e.g. keeping separate copies of received emails, then the platform is fundamentally insecure which is out of scope. However, a server attack may occur after the server is discarded by physically compromising the server's disks [42]; few organizations erase old disks before disposal. We assume the enemy is sophisticated but not at the level of an intelligence agency, i.e., the enemy cannot break TLS.

We do not attempt to protect against compromise of the user's devices or mail clients. If those are compromised, the private keys used by E3 are available to the attacker no matter when the encryption takes place. Standard end-to-end encrypted email makes the same assumption.

3.3 Usage Model

E3 works with any IMAP email service. To get started, a user installs an E3 client that is either a separate app or a full mail client. The latter may support E3 natively or via an add-on. E3's setup is similar to a normal mail client which asks for the user's email service and its credentials. If we assume a user uses only one device to access email, then once that device's E3 mail client is setup, the client will begin encrypting all email on receipt. The user then continues using whatever email client he wants exactly as before, including sending and receiving email, except that the E3 client transparently encrypts emails on receipt. Mail clients that support encrypted emails identify them

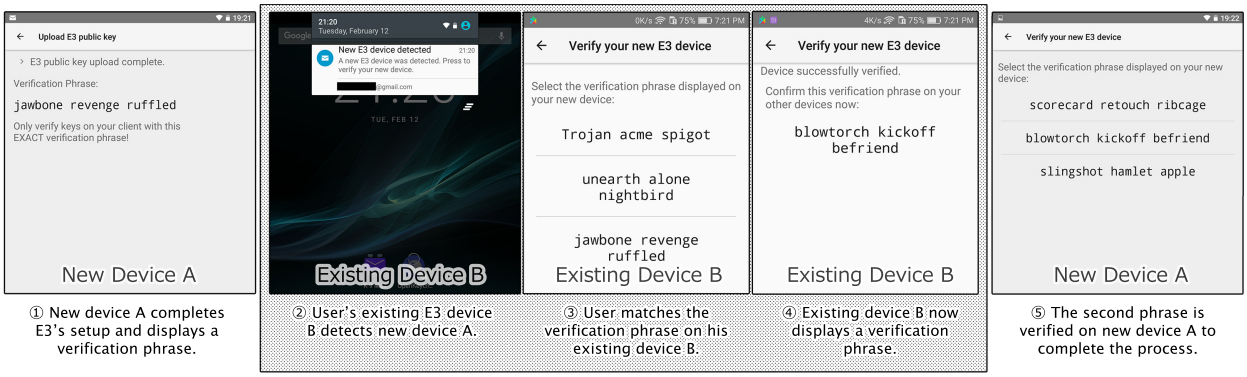

Figure 3.1: E3's two-way verification process.

with visual indicators to avoid being *too* transparent [18] as it should be obvious whether an email was encrypted or not.

Modern email users often use multiple devices to access email, and E3 is specifically designed for and encourages users to configure multiple devices with E3. To do so, users participate in a simple, brief, and platform-independent two-way verification process for each new device as summarized in Figure 3.1. Suppose that a user's initial E3 client is on his smartphone, and now he wants to configure E3 on his laptop computer. The user wants the smartphone and laptop clients to trust each other with email access, so they participate in a two-way verification. The user performs the E3 client setup process on the laptop, except it will show the user a verification phrase at the end. The smartphone client detects the laptop's client and prompts the user with a choice of several phrases on the smartphone and asks the user to select the one that was displayed on the laptop. After selecting the correct one, the user repeats to process with the laptop and smartphone swapped; the smartphone displays a verification phrase which the user must select correctly on the laptop. This completes the two-way verification, and the smartphone and laptop now trust updates from one another. If the user wishes to add a third device, say a tablet, he performs the two-way verification process with any of the previously configured devices. If he verifies the tablet on his smartphone, then his laptop can transitively trust the tablet via the phone.

If the user does not select the correct verification phrase within a time limit, the verification process is canceled and the user will need to restart the E3 verification process on the new device.

When the user succeeds in verifying a new device, the user is informed that it will take some time for any previously encrypted emails to become readable on the new device. The reason is because these emails need to be re-encrypted so that the new device can read them.

Importantly, users rarely set up new devices or mail clients, so re-encrypting emails is an uncommon cost. Adding a new device generally happens in the following situations: (1) a user replaces an existing device, or (2) a user obtains an entirely new device. If a replacement, then in many cases the old device's data is cloned to the new device so that neither verification nor re-encryption is necessary. Case (2) is an uncommon occurrence, but a new device means the user will need to verify it and re-encrypt emails; however, any future replacements will fall under case (1).

When a new device is added, the clients on all previously added devices display a notice to the user that his emails are being re-encrypted. The user has the option to cancel this process and return the emails to their original state. Upon cancellation, the client rolls back the work it thus far completed. Similar logic is applied if the user wishes to revoke a device from his E3 ecosystem. A user removes a device by deleting it by name from any device configured with E3. The remaining clients then re-encrypt all email to exclude the deleted device.

A user may occasionally no longer be able to use a device, because it has been damaged or is no longer operational. If the user has multiple E3 clients as would commonly be the case for users that have multiple devices to access email, he can still access his E3 encrypted email on his remaining device(s). If the user only had one device with E3, a backup of the old device's data can be simply cloned to a replacement device to regain access to email on the replacement device. With mobile devices which are more easily damaged, backups are increasingly common. If it is desired to support users who use only one E3 mail client device that is never backed up, E3 can provide the user with a recovery password which the user must save by printing it out or recording it somewhere safe. Users use the recovery password on a new E3 client to access to their emails. No recovery password is needed for users who use multiple devices unless they fear they may lose all of them simultaneously.

While most mail clients support encrypted email, one exception is web browser clients such as the Gmail website and other webmail services. Browser add-ons for encrypted mail exist (and we have also written one), and it is reasonable to expect native browser support if the demand is great enough. For now, web browser extensions can integrate with webmail services to decrypt E3-encrypted email.

E3 assumes that email should only be accessible from trusted devices. Given the ubiquity of mobile devices and that most users use them for accessing email [43], this assumption is quite reasonable for modern users. E3 is not compatible with using untrusted computers such as those at an Internet cafe, nor should it be if users care about their email privacy given that such computers may be compromised. Attempts to use such untrusted computers to read email will not work; they will only provide access to encrypted emails.

3.4 Architecture

The E3 architecture consists of two main components, an encrypt on receipt mechanism and a Per-Device Keys (PDK) architecture. For simplicity, we first describe the encrypt on receipt mechanism using one E3-enabled device, then describe how multiple devices are supported using the PDK architecture.

Figure 3.2 presents a high-level view of E3's encrypt on receipt mechanism. An E3 mail client downloads an email, encrypts it in either PGP or S/MIME format using a self-generated keypair or X.509 certificate, and uploads the encrypted version while deleting the original. For ease of discussion we refer to PGP keys and X.509 certificates as keypairs consisting of public and private keys. E3 builds on existing protocols and encrypted email formats, simplifying its implementation and deployment. E3 leverages Internet Message Access Protocol (IMAP) [44]. S/MIME implementations rely on X.509 certificates and the S/MIME standard as documented in RFC 5280 [45] and RFC 5751 [46]. PGP implementations follow the OpenPGP standard in RFC 4880 [40].

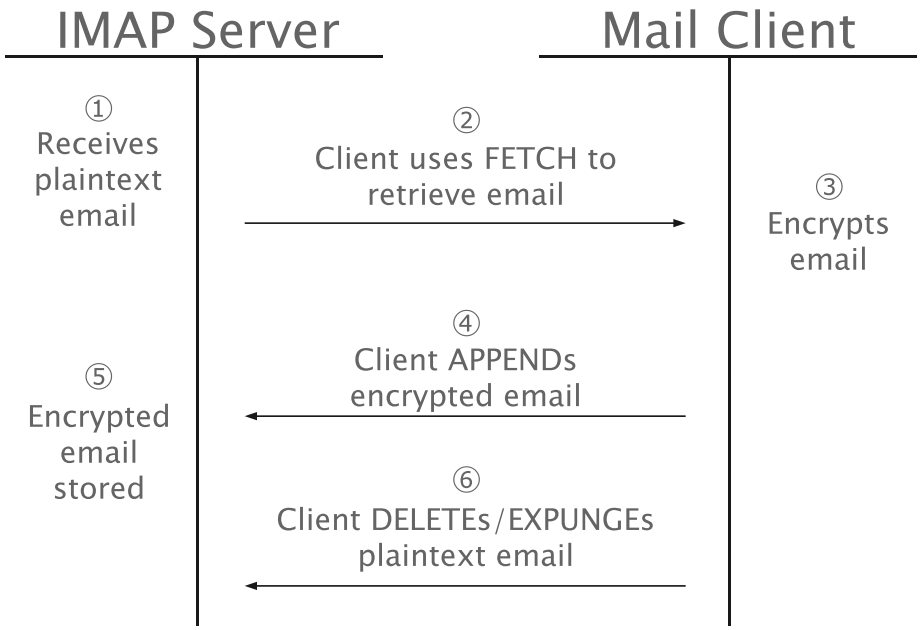

Figure 3.2: Communications between an E3 mail client and an IMAP server to encrypt email.

3.4.1 Keypairs without PKI

Normally, public keys need to be signed by a trustworthy entity or nobody will trust it. This forms the basis of PGP webs of trust and PKI. PDK public keys are never shared with other people. They are self-generated and self-signed, and require no PKI for the user to understand. Previous work [13] has shown that users find it confusing to correctly obtain and use public keys. In contrast, an E3 user needs only self-generated keys, and any public key exchanges among his devices are automated.

3.4.2 IMAP Support and Compatibility

Consider common email operations. A mail client downloads a message using the IMAP FETCH command. To delete it, the client uses the IMAP STORE command to mark it with the `\Deleted` flag. IMAP EXPUNGE then purges email marked for deletion. The user may compose and upload an email using IMAP APPEND. These four IMAP commands, FETCH, APPEND, STORE with `\Deleted` flag, and EXPUNGE, play a key role in E3. We henceforth use

DELETE as shorthand for the STORE with `\Deleted` flag command.

Figure 3.2 shows how these four IMAP commands encrypt email on receipt with existing IMAP servers. E3 is summarized as downloading a message (FETCH), encrypting it, uploading the ciphertext (APPEND), and deleting the cleartext (DELETE and EXPUNGE). Finally, the client ensures correctness by synchronizing with the server.

This series of commands works on any IMAP message. It does not matter what mailbox or folder the message is in. The same process is even applied to a user's copies of his sent emails which are appended to the IMAP server (these appear as "Sent" emails to users). All these IMAP commands execute in the background, decoupling them from the critical path of reading email.

E3 requires multiple round-trip times (RTTs) with the server because IMAP does not support message replacement. Optimizations may be possible in the future. The proposed REPLACE command [47] substitutes for the APPEND, DELETE, and EXPUNGE commands. Although this RFC extension has been elevated to an IETF Proposed Standard in 2019, it is not yet adopted by major IMAP mail services. The REPLACE command would eliminate the multiple RTTs associated with DELETE and EXPUNGE thereby significantly improving performance when replacing many small emails. This is because RTTs have a constant cost that dominates the brief time it takes to encrypt and replace small emails. In contrast, the RTTs account for a small percentage of the total time for processing large emails and are unnoticeable. Another optimization would be to use IMAP pipelining, but not all IMAP servers support it, and REPLACE would obviate the need for it.

E3 is compatible with TLS [48] (or STARTTLS) which encrypts all communications with the IMAP server. Although eavesdropping is not E3's primary security focus, E3 with TLS protects against attackers who could otherwise capture cleartext emails when they are first downloaded by the client.

E3 uses approaches similar to existing IMAP clients in dealing with race conditions since multiple clients may try to encrypt the same message which could result in duplicated encrypted emails. Currently, the preferred way of achieving pseudo-atomicity when modifying IMAP mes-

sages is to use the IMAP CONDSTORE extension [49]. CONDSTORE is supported by major IMAP email services and open source servers, including Gmail and Dovecot. This extension requires servers to maintain a last-modified sequence (mod-sequence) number on messages which is returned to the client. An E3 client which wishes to encrypt a message adds a flag (either `\Flagged` or `\E3Encrypting` depending on custom flag support) to a message using the UNCHANGEDSINCE modifier with the IMAP STORE command so that it will only succeed if the message has been unchanged; this also updates the mod-sequence value of the message, so any other clients who try to issue the same command will fail since the message was already modified.

The flag and mod-sequence value act like a lock, thereby alerting other clients that this message is being encrypted. Then, the client with the lock can issue IMAP commands without racing others. One issue is the client may crash before it completes its work and leave a dangling lock. A basic solution is to use a heuristic based on a message's received timestamp. A client periodically scans the mailbox for messages with the `\E3Encrypting` flag, and based on the timestamp heuristic, determines if too much time has passed since each message was received. For example, if a message is unencrypted for three hours since it was received but has the `\E3Encrypting` flag, the client may obtain the lock on the message and encrypt it.

If CONDSTORE is not available, an alternative is to make a best-effort using IMAP custom flags and custom IMAP folders. The strategy, like with CONDSTORE, is to mark a message with a custom flag (keyword) entitled `\E3Encrypting`, and to move it into an IMAP folder named `E3-Temp`. Then, any E3 client that sees the E3 flag on a message in the special temporary folder should not encrypt it. This does not rule out race conditions entirely, but will certainly shrink the window that it could occur within.

3.4.3 Ciphertext Format

E3 uses the widely supported OpenPGP message or S/MIME Enveloped-Data formats depending on client preference. While E3 can be implemented as a full standalone mail client, it can also be implemented as a program that provides just the encrypt on receipt mechanism. Users can then

use existing unmodified mail clients that support S/MIME, including Apple Mail, Mozilla Thunderbird, and Microsoft Outlook, to access E3 mail in S/MIME format, assuming the E3 private key is available on the device to both the encryption program and the existing mail client. The same holds for PGP.

These formats only encrypt the body text, so all of the original headers are maintained except for the Content-* headers which are updated to ones appropriate for encrypted emails. Since the Received timestamp header is unchanged, mail clients can display messages in their original order. E3 also adds a custom header, X-E3-ENCRYPTED, to distinguish E3 emails from other encrypted emails. This is useful for IMAP servers which do not support custom flags or keywords.

E3 normally does not re-encrypt emails that are already encrypted when received, i.e., when receiving email from a sender using end-to-end encryption. However, there are situations where re-encrypting emails is useful such as when a crypto algorithm or key size is no longer secure. In this case, E3 supports re-encrypting existing encrypted email to a newer crypto standard.

E3's encryption does not interfere with spam filters. Spam filters often exist either on servers or clients. When they are on the server, such as with Gmail, the mail service filters spam emails before they are encrypted. For client-side spam filters, the user's mail client will detect spam messages and move or delete them. However, since the client performs the filtering, it can apply the filter before encryption, or decrypt E3-encrypted messages to scan them for spam.

3.4.4 Search Capability

Searching is straightforward: index and store the decrypted content of messages locally. This is compatible with existing mail client local search, and provides full, fast local searching. Storing messages locally is a common practice among modern mail clients, and examples can be seen in Gmail on Android, Mail on iOS, and Mozilla Thunderbird and Apple Mail on desktops. While message content is stored locally in the clear, many mail clients that support encryption already do this. An option for the more security-conscious is to apply full disk encryption in conjunction with device-wide security features. An alternative is to store ciphertexts locally, but this provides

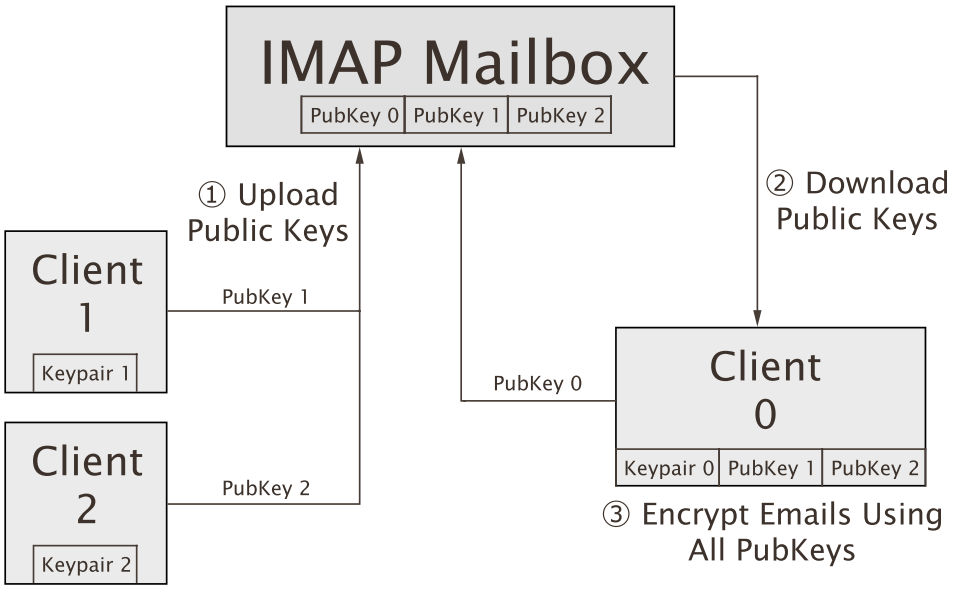

Figure 3.3: The PDK architecture for E3.

no real benefits since the key is also stored locally, and would also interfere with local searching.

A limitation of encrypted email schemes is that unmodified email servers cannot search the body content of encrypted emails. (Headers, including the `Subject:` and other metadata fields, are searchable.) If the server can be modified, SSARES [50] is a scheme for searchable encrypted email without access to private keys, making it compatible with E3's threat model. For unmodified servers, the IMAP SEARCH command cannot be used, so clients that search both locally and on IMAP servers will only return results for local search and remote metadata matches. On the other hand, IMAP search is significantly slower than local search and is often based on naive string matching which yields low quality results. Thus, users often will not wait for IMAP server search results in practice since local search queries are nearly instant. Furthermore, many email clients such as K-9 Mail only perform local search unless remote search is specifically enabled or requested, which would provide the exact same search capability with or without E3.

3.4.5 Key Management, Migration, and Recovery

E3 eliminates manual public key exchanges through its use of PDK. This simplifies the key

management by removing half of it. What remains is the problem of private keys when using multiple devices. As mentioned in Chapter 2, PDK returns to the traditional security advice of never transporting private keys. In contrast to most secure email schemes which assume a user has a single private key to be moved or copied to multiple devices, E3 users have a unique private key for every device; Figure 3.3 depicts E3's instantiation of the general PDK design. A user does not need to move any private keys among his devices. Instead, each of his clients automatically makes available its *public key* to his other devices. The result is that any E3 client can then encrypt the user's emails using the public keys from all of his devices. Consequently, any of a user's multiple E3 clients can encrypt emails while making them readable on any other client. The principle is similar to when a traditional PGP or S/MIME user encrypts an email to multiple people. The email is not encrypted multiple times for each public key, but is encrypted only once using a symmetric key which in turn is encrypted to each public key. E3 takes this paradigm and applies it in a new way in the PDK framework by encrypting emails on receipt using every verified public key belonging to the user. When a new key is added, clients re-encrypt already-encrypted emails to the new keys.

The primary features of PDK are listed in Chapter 2, but E3's take on PDK includes the following distinctions:

(2) A private key is "revoked" by re-encrypting emails to all public keys except for the revoked one.

(4) Public keys are automatically distributed using the user's email account as the communication channel.

E3 clients upload their public keys to the mailbox as ordinary emails with the keys as attachments. Other E3 clients detect these key emails and store the public keys locally. Table 3.1 shows custom MIME headers used in E3 key emails to support PDK. Invalid or missing headers (when they are required) cause a key email to be rejected immediately. The concert of these headers is used to support key verification.

Header	Description
X-E3-NAME	A custom name for this E3 public key.
X-E3-VERIFICATION	The verification phrase as a space-separated string.
X-E3-TIMESTAMP	The signed timestamp of when this key was uploaded.
X-E3-DIGEST	The digest (fingerprint) of this E3 key.
X-E3-RESPONSE	The digest of the key that this key is in response to.
X-E3-KEYS	The public keys known to the uploader.
X-E3-DELETE	The public key deleted from the uploader.
X-E3-SIGNATURE	Signature of all fields using the uploader's private key.

Table 3.1: Custom headers in uploaded E3 key emails.

Public keys in the user's mailbox cannot be blindly trusted. Clients must securely confirm whether a new PDK public key really belongs to the user, and ideally, the method to do so should be compatible with any kind of device whether a desktop or mobile one. The following solution satisfies these requirements. A given client periodically scans for new keys, and as a first heuristic, ensures that the sender (i.e., the "From:" field) of any detected key email matches the address of the account owner; any emails containing keys from other senders are not accepted. However, this heuristic alone is not enough to verify the key as an attacker may spoof this field or gain access to the email account and upload a malicious key with the correct sender address. We therefore augment this check by requiring temporal proximity and a two-way verification step. Temporal proximity means the user has a limited window of time to accept and verify a newly detected public key. Any keys which are not accepted within the time window will expire.

Temporal proximity relies on verified and signed timestamps. A newly configured client uploads its public key along with a signed timestamp obtained from services such as Roughtime [51], and existing clients verify if the timestamp is within the allowed time window and trustworthy. For example, a client configured to only allow public keys uploaded within the last 60 seconds will reject any uploaded public keys with a verified timestamp that is older than 60 seconds. The timestamp is verifiable since it is signed using the Roughtime service's certificate. As an additional measure, clients rate limit the number of requests to add a new key. For example, the client will only consider at most three key requests in a period of five minutes. Any more than that are suppressed, and a warning is shown to the user that unusual behavior has been detected.

E3's PDK also uses a two-way verification process with a verification phrase that is easy for humans to recognize and match. When a new client uploads its key, it adds a randomly generated verification phrase to the key email which is prominently displayed. The user then needs to confirm this verification phrase on one of his existing E3 clients. Once he completes the verification on any existing client, it will display a second verification phrase. The user then needs to confirm this second phrase on his new client to complete the two-way verification.

The catch is that when the user confirms a verification phrase, it must be selected from among two randomly generated incorrect phrases. The user must select the correct verification phrase in order to verify the key. This multiple choice confirmation reduces the chances of a user accidentally accepting a key that isn't his. The words in the phrases are selected from a curated pool such as the PGP Word List [52]. As shown in [53], this technique is effective and usable for quickly authenticating identities even with only three words. Users who speak other languages use word lists in their language. Another option is to use a recognizable but randomly selected or generated image. Further research is needed to better understand what kinds of strings or images real users can correctly recall and verify while making minimal errors.

To concretely visualize how adding and deleting E3 clients works, we will describe the one device, two device, three device, and n device cases for E3's PDK design. To represent uploaded key emails, we use the notation $KeyEmail_d(Key_d, \{h\})$ where d is the device which uploaded the key email, Key_d is the public key of the uploader, and $\{h\}$ can be any of the values shown in Table 3.1 with X-E3- removed for spacing reasons; we also elide the required VERIFICATION and TIMESTAMP headers but note that they are necessary in each $KeyEmail$ we describe. To denote what public keys a given device d knows about, we use $d[Key_{d0}, Key_{d1}, ..., Key_{dn}]$.

One Device. Since there are no devices to synchronize keys with, a user simply sets up an E3 client on his single device and begins encrypting emails on receipt.

Two Devices. Let us consider two devices, A and B, where A is a device with E3 already configured on it, and a user wants to add B to his E3 ecosystem. Thus, the initial state of knowledge is $A[Key_A]$ and $B[Key_B]$. The user sets up an E3 client on B and it uploads $KeyEmail_B(Key_B, \{\})$,

then shows the user a verification phrase. Now device A detects $KeyEmail_B$ and requests the user to verify it with the phrase shown on device B. If the user succeeds, device A now knows device B's public key, but since $KeyEmail_B$ did not contain X-E3-RESPONSE, device A knows it needs to upload its own set of keys so that device B can learn about existing public keys. Device A therefore uploads

$$KeyEmail_A(Key_A, \{\text{RESPONSE}[Key_B], \text{KEYS}[Key_A, Key_B], \text{SIGNATURE}_A\}).$$

A then displays its own verification phrase to the user, which he must verify on device B after it detects $KeyEmail_A$. If this second verification succeeds, now both devices A and B know about their public keys and can trust future updates from each other. The final state is $A[Key_A, Key_B]$ and $B[Key_B, Key_A]$.

Three Devices. The same process for two devices holds for adding a new third device C because A and B trust each other, so if C is verified and added to A, A will upload

$$KeyEmail_A(Key_A, \{\text{RESPONSE}[Key_C], \text{KEYS}[Key_A, Key_B, Key_C], \text{SIGNATURE}_A\})$$

which B trusts because of the signature, so B can automatically add Key_C to itself. Then once the user does the response verification of A on C, C will trust A as well and can add Key_B. So the final state is $A[Key_A, Key_B, Key_C]$, $B[Key_B, Key_A, Key_C]$ and $C[Key_C, Key_A, Key_B]$.

N Devices. Now consider a user who has built up his E3 ecosystem over time and has $N-1$ devices already synchronized with each other, and now he wishes to add device N. The user completes the two-way verification process with device N and any device K in $0, ..., N-1$. Then K automatically distributes N's public key to every other device by leveraging transitive trust because the other devices already trust K. Since K is manually verified on device N by the user, N can trust the keys that K provides.

Clients must re-encrypt all emails for new public keys, but a user may wish to undo adding a new device. If the user stops and reverses the re-encryption process, the client re-processes

the emails it re-encrypted, and re-encrypts them again to the original set of keys. However, the now-defunct key must be revoked first.

The general case of revocation is done via an advertised deletion. When a user revokes a client, he deletes the key by name from any client's list of keys. The client which performs the deletion announces this by uploading a signed key email with the X-E3-DELETE header so that other clients can also exclude the revoked one.

E3's approach to PDK achieves a streamlined key verification process where, for every newly added key, the user only ensures that the verification phrase matches the one he recognizes two times. This is in contrast to key verification for end-to-end encrypted email which often relies on confusing public key fingerprint matching, QR code scanning which is unavailable without a camera, and understanding of PKI. Although E3 keys can be verified with these techniques, the higher guarantee (and difficulty) they provide is unnecessary given the unique environment in which E3 operates. Another issue with end-to-end encrypted email is verifying the public key of every new email correspondent. In E3, adding a new PDK key is a rare occurrence and only happens when configuring a new mail client such as when getting a new device. As a side note, advanced users may prefer key fingerprint matching or QR code scanning. These are only available as an advanced option that is not enabled by default.

E3's recovery mechanism inherent to the PDK multi-device design is available to the majority of users who access email using two or more devices. However, there may be users who truly only ever access email with a single device. As discussed in Section 3.3, for these users, a backup of the old device's data can be simply cloned to a replacement device to regain access to email on it. For users with only one E3 mail client device that is never backed up, PDK key recovery uses the traditional method of encrypting the user's private key with a password, presented as a "recovery key" to users, and then storing the encrypted private key on a backup device or in cloud storage. If stored in cloud storage, the provider should be different from the email service provider. For example, E3 clients configured for Google's Gmail service might store the private key on Dropbox but not Google Drive.

E3 is also compatible with re-encrypting emails to future-proof them against changes in crypto standards. Algorithms age, so ciphers and key sizes that are secure today may not be in the future. PDK supports this use case since a user can generate and add new keys while deleting old keys at will.

One avenue for future work is the problem of reading email on public computers. In this case, users access their confidential data on a fundamentally untrusted device which cannot be trusted with private keys. This is a concern for all encrypted email schemes, not just E3. Even though solutions are technically possible, they are insecure due to the high risk of unwrapping private keys in an untrusted environment.

3.4.6 E3 Configurations

While we have assumed that all E3 clients encrypt on receipt and perform PDK, it is possible to configure E3 clients to only generate a PDK keypair and perform no encryption of its own. Note that at least one E3 client needs to encrypt on receipt to protect a user's email. Clients which only generate a PDK keypair configure a user's device to decrypt emails and do not encrypt emails. An example is a one-time use app or add-on which configures a user's existing, unmodified mail client with an E3 private key. These clients *only* perform the key management functionality described in Section 3.4.5 and none of the encryption, and are a strict subset of E3 clients which do encrypt.

3.5 Security Analysis

E3 does not intend to be an end-to-end, maximum security solution, but a strict improvement over the norm that is easy to use and deploy. We sacrifice a small amount of security to gain tremendous usability over existing secure email models. We henceforth show that E3 provides tangible security benefits compared to no email encryption, and compare its security with traditional end-to-end secure email.

E3 protects all emails for all of their lifetime as long as they are encrypted *before* any email account or server compromise. Standard end-to-end encryption does the same, but E3 does so

without the complexity of public key exchanges and PKI.

Like end-to-end encrypted email, E3 protects sent and received mail assuming all correspondents use E3. Senders can encrypt their sent email copies as stored on their IMAP server. Unlike end-to-end encryption, which *requires* that both the sender and receiver use it, E3 provides useful protection even if only one side uses it. If the sender uses it, his emails that are encrypted before an attack are protected from compromise of his email account or server. The same holds for the receiver without loss of generality. In other words, E3 provides better protection than end-to-end encrypted email for communications in which one party does not use email encryption because end-to-end encryption cannot be used and would therefore provide no protection at all.

If not all email correspondents use E3, it is possible for an attacker to compromise the emails of any correspondent not using E3 to expose email communications with one that uses E3. Regardless, this property actually confers a benefit to E3. E3 can be incrementally deployed since not all correspondents require it. E3 also exhibits network effects: it provides better security as more users use it.

Unlike end-to-end encrypted email, E3 requires additional measures to protect against eavesdropping. Fortunately, these measures are completely transparent to users. E3 uses TLS or STARTTLS so there is no threat of eavesdropping if TLS is secure. Furthermore, TLS and STARTTLS are supported and encouraged by practically all major mail services.

Email may or may not be protected in transit between SMTP (not IMAP) servers. SMTP server links are increasingly protected by TLS; if not, the problem is out of scope. Services such as Gmail flag emails that arrive via unprotected SMTP connections. That said, attackers tapping such backbone links is out of scope for E3 and in general is difficult for any party but an intelligence agency.

After an email account or server is compromised, E3 cannot protect newly arriving emails. This is a limitation compared to end-to-end encryption which protects new emails assuming all email correspondents use it. Nevertheless, end-to-end encryption rarely sees actual use among users and therefore provides no practical security for the majority of the population. In contrast, E3's ease of

use makes it much more likely to be adopted while providing a strict security benefit. In a mailbox with just a few thousand messages, compromise of new emails is a minuscule percentage of total emails. New emails are important, but it is clear that encrypting the majority of emails is better than none.

Email account compromise happens in many ways but it is primarily through credential or key compromise. That, in turn, often happens because of user error, especially in cases of (spear-)phishing. While devices do have OS-level security features to help combat phishing, E3 by design also provides a strong defense even though it does not password-protect private keys since the device is assumed to be secure (it is better to rely on OS level protections such as seen in Apple Mail and Autocrypt [54], and also there is now no password for an attacker to phish). The critical aspect is that E3 makes informed decisions about private key storage and management based on the user's platform and device, so users are never requested to manage their private keys in contrast to PGP and S/MIME which require a user to actively manage and move around a private key. Thus, (non-technical) users have no knowledge of where the E3 private key is stored. This latter intrinsic property of E3 also raises the bar for an attacker to trick a user into providing his private key since the user does not know where it is. Attackers would therefore need to provide detailed instructions unique to platform and device for users to find the private key.

One major obstacle in other secure email schemes is ensuring availability of the private key on all devices. There is no standard for secure, usable key transport and the market is fragmented. In general, most solutions assume that a user has a single keypair which is either copied to all his devices, or carried on his person such as on a security token or USB device. We have designed PDK as a departure from these approaches. It provides a secure and usable scheme that leverages users' tendency to access email on multiple devices, and also the inherent support for multiple recipients in encrypted email formats.

An attacker may try to trick a user into accepting and authenticating a malicious public key by sending a fake E3 key email to the user. If the user were to accept it, all emails would be encrypted using the malicious key, allowing the attacker to decrypt the user's email if the account

is ever compromised. Therefore, PDK is only as strong as the key authentication system used in conjunction with it. The first line of defense is to ensure that uploaded keys came from the user's own email address. Keys attached to email from other addresses are rejected. However, an attacker may spoof the sender address or have access to the email account allowing him to craft legitimate emails with the correct sender. We therefore rely on temporal proximity such that an attacker would need to strike literally minutes or even seconds before the user generates a new key. Otherwise, the uploaded key would be rejected for being too old if encountered by the target at a later time. This is similar to time-based one-time password schemes as seen in two-factor authentication, e.g., RSA security tokens and Google Authenticator.

An attacker without access to the mailbox needs to also guess the correct verification phrase. An attacker with access to the mailbox could wait for the user to upload a new key, duplicate the key email but attach his malicious key instead, and delete the real key email. This would allow the attacker to construct a key email with the correct verification phrase, and this may go unnoticed by the user and his other E3 clients. However, this attack requires immediate temporal proximity, i.e., as soon as the user uploads a new key, and moreover, the client that performed the key upload can detect this attack even if other clients cannot. To do this, the uploading client polls the server to see if the key email it uploaded was deleted or moved, or if another key email with the same phrase was uploaded. The client can distinguish the real email from a fake one in any case simply by referencing the real key email's IMAP UID which is generated by the IMAP server, not the client. As soon as the client detects an issue, it warns the user that an attack may be occurring.

Another possible attack is to try to exploit E3's automatic public key distribution approach by either trying to propagate a malicious key to valid clients, or trying to delete valid clients. To propagate a malicious key addition or deletion, an adversary could upload a fake key email for either case. A fake key addition email would not be verified by a user, and thus the attack would fail. A fake key deletion email would not be accepted by any valid clients because the signature (X-E3-SIGNATURE) would be incorrect.

An adversary may resort to a denial-of-service attack and send many fake keys to a user in

hopes the user will make a mistake and accidentally verify a malicious key. To address this, clients rate limit requests to add new keys and show a warning to the user. As a final measure, clients also immediately discard keys and any on-going confirmation prompts from any key emails with duplicated verification phrases.

These checks alone suffice to exclude most attacks. On top of these key verification checks unique to E3, we *optionally* support traditional methods for verifying public keys including fingerprint string matching and QR code-based fingerprint verification. However, these methods are only be available to advanced users and are not enabled by default.

E3 considers servers and devices that are malicious from the beginning as out of scope. E3 cannot protect against an IMAP server that is run by a dishonest service provider. This then begs the question of whether popular email services can be trusted. As a case study, Google's retention policy [55] states that when a user requests a deletion, Google immediately begins deleting that data from all its systems, but it may take some time for the data to be completely removed from every internal Google server. At the least, the data is no longer accessible from user-facing interfaces such as Gmail thus preventing any external adversaries from gaining access to deleted emails. Google clearly states that it does delete data completely, so if it were to do otherwise, it would be subject to US law [56, 57] which prohibits "deceptive practices" by any entity engaging in commerce. Similar laws apply in other regions as well.

E3 also does not protect against compromise of the user's devices or mail clients, but neither does end-to-end encrypted email. Similarly, if a user's device is stolen, E3 cannot protect his email. However, many devices are password-protected with data encrypted in local storage, and have remote wipe functionality. In all cases, E3 provides a strict security benefit, and makes security no worse than the current common practice of no email encryption.

3.6 Implementation

To demonstrate that E3 is easy to implement, we built four different E3 prototypes for various platforms: a K-9 S/MIME client, a K-9 PGP client, a Python encryption client, and a Google

Chrome extension.

We implemented E3 in K-9 Mail, a popular open-source Android mail client, using S/MIME. K-9 Mail's developers by design include no crypto libraries and offload crypto to separate crypto provider applications. However, K-9 has no S/MIME support since no such provider for S/MIME currently exists. Our K-9 S/MIME implementation therefore includes the Spongy Castle [58] crypto library and performs all key generation and management on its own. K-9 S/MIME represents a worst case scenario where nearly email crypto functionality is implemented from scratch. Excluding third party libraries such as Spongy Castle, which adds 8.6K lines of code (LOC), our K-9 S/MIME implementation only added roughly 2.5K LOC. The entire K-9 codebase is around 210K LOC, excluding XML code which adds another 200K LOC, thus suggesting that E3 comparatively represents a modest amount of complexity. We also implemented a more optimized E3 K-9 S/MIME version by replacing Spongy Castle with precompiled OpenSSL libraries to leverage hardware encryption support on Android devices. While there was no change to the E3 code needed, the OpenSSL libraries are substantially larger than Spongy Castle, roughly 390K LOC. Although Android includes its own OpenSSL as a system library, the version included is heavily modified and strips many features including the S/MIME functions required for our implementation.

We also implemented E3 in K-9 Mail using PGP by relying on the OpenKeychain Android app, which is both a keychain and crypto provider. K-9 offloads all PGP and key operations to OpenKeychain which exposes an external cross-application API, so it was not necessary to add a crypto library to K-9. We modified K-9 Mail and OpenKeychain to support E3. We added an API call to OpenKeychain (OpenPGP-API) for storing E3 keys, and changes to make OpenKeychain verify and recognize emails which have been self-signed by the email recipient as opposed to the standard PGP use case where it verifies signatures based on the email sender. Our E3 K-9 PGP client had nicer UI features compared to the K-9 S/MIME client, adding 3.3K LOC, with much of the additions being UI boilerplate code. Our changes to OpenKeychain were about 250 LOC, while the entire OpenKeychain codebase is 590K LOC, excluding 124K LOC of XML. Without

the need for additional crypto libraries, the total amount of additional code to support E3 was only 3.6K LOC out of the over a million LOC required for K-9 and OpenKeychain.

We developed a Python E3 daemon for Windows, Linux, and macOS that generates an E3 keypair and encrypts on receipt, but does not currently automatically add the private key for use with existing mail clients; users must manually perform this step, so the daemon is currently intended for use by more technical users. The implementation is only 1K LOC. We sketch out what automatically adding the key to mail clients would look like on different platforms. On macOS and iOS, we can leverage the system Keychain which the Apple Mail and iOS Mail clients already integrate with. The Python app can add its E3 keypair to the Keychain with an ACL tailored for the targeted mail clients [59]. On Android, the KeyChain API [60] stores system-wide keypairs and can be used in a manner similar to Apple's Keychain. However, Android clients that do not rely on the KeyChain API will require modifications; for example, OpenKeychain must be modified to allow an app to add an E3 private key to it, then existing PGP clients can seamlessly use the key. On Windows, the E3 client can generate a PKCS12 key file to import into Windows' certificate store which is used by the Outlook mail client. For clients such as Mozilla Thunderbird that do not rely on the certificate store, users can install an E3 add-on.

We prototyped a Google Chrome extension to interface with the Gmail website and support reading E3 encrypted emails and key management. This extension was a proof of concept to show that reading E3 email on web mail clients is possible and practical, but does not perform encrypt on receipt. It is about 750 LOC plus 7.5K LOC for external Javascript libraries for crypto and other important functionality. The extension requests access to the user's Gmail API to process raw emails instead of scraping Gmail's DOM. When a user loads an encrypted email in Gmail, the extension checks if it can be decrypted, fetches the email, decrypts it, and injects its contents into the page. The extension uses the Gmail API to also perform the necessary key management functionality for E3. However, Google Chrome by design provides no secure storage whether for extension data or browser cookie data. It instead relies on its own and OS security features to protect sensitive data. Thus, we store the E3 keypair in Chrome's local storage.

	Gmail	Yahoo	Outlook	AOL	Yandex	Dovecot
E3	○	○	○	○	○	○
CONDSTORE	○					○
REPLACE						

Table 3.2: Tested servers and their compatibility with E3.

3.7 Experimental Results

We verify that E3 works with existing IMAP services, measure its performance overhead, and evaluate its usability with real users. We used K-9 S/MIME for performance testing, and K-9 PGP for usability testing.

3.7.1 Compatibility and Interoperability

To verify that E3 is compatible with existing IMAP and S/MIME systems, we tested our prototypes on several of the most popular commercial and open-source email servers. Table 3.2 shows the results of our compatibility testing. E3 worked seamlessly with all IMAP email services tested. We also checked for IMAP CONDSTORE and REPLACE support with the former enabling better IMAP atomicity, and the latter enabling better performance. We also verified that unmodified S/MIME mail clients, including Apple Mail, and Thunderbird, could be used to read E3-encrypted email.

3.7.2 Performance

We measured E3's performance on mobile devices because of the popularity of mobile email and to provide a conservative measure as they are resource constrained. We used a Huawei Honor 5X (8-core Cortex-A53 with 2 GB RAM) smartphone running Android 6.0.1. We compare the performance of our E3 K-9 S/MIME client against the standard K-9 Mail client. Both versions were instrumented to obtain measurements. The E3 K-9 client used OpenSSL 1.1.0b, and the S/MIME emails used Cryptographic Message Syntax (CMS) with 128-bit AES CBC for compat-

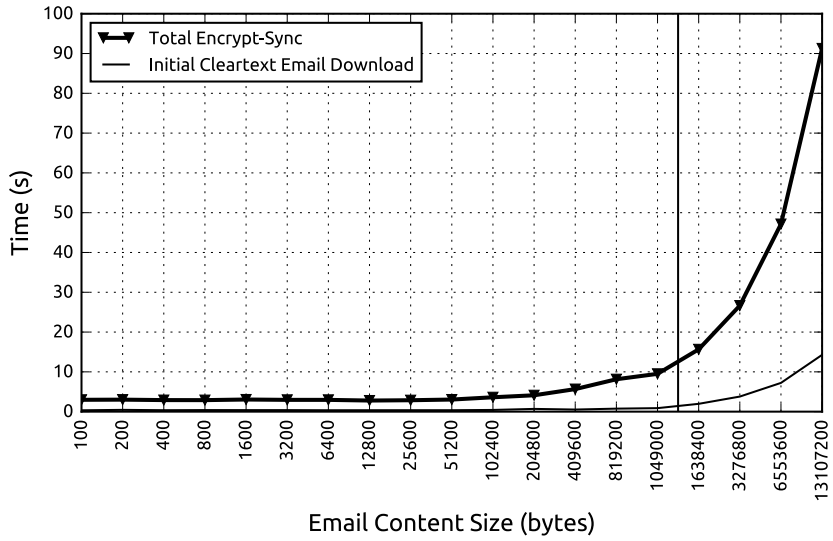

Figure 3.4: Time spent for the one-time "encrypt/synchronize" compared to the cleartext download. Points right of the line are emails with a JPEG.

ibility reasons. All experiments were conducted using Gmail accounts populated with the same email content, and a WiFi connection to a small business fiber optic network. We chose to use a real email service with a typical Internet connection to better understand performance with real limitations, such as asymmetrical download/upload speeds to the Gmail service. To account for variability, each measurement was repeated 30 times, the three lowest and highest outliers were discarded, and an average was taken over the remaining measurements.

We considered email operations where E3 imposes additional work over a standard email client. We did not measure searching as it has no overhead compared to a standard mail client. We measured receiving a new cleartext email in which E3 downloads, encrypts, and replaces it at the server with the encrypted version, followed by a quick synchronize.

We used a range of email content sizes from 100 B to 12.5 MB. 12.5 MB is the maximum because when encrypted, it increases in size to about 24.7 MB due to limitations of the MIME format. Popular services such as Gmail enforce a 25 MB size limit. Emails of size 100 B to 1 MB were two-part MIME messages with a `plain/text` and `html/text` part. Larger emails were two-part MIME messages with a one byte `plain/text` part, and an attached JPEG file.

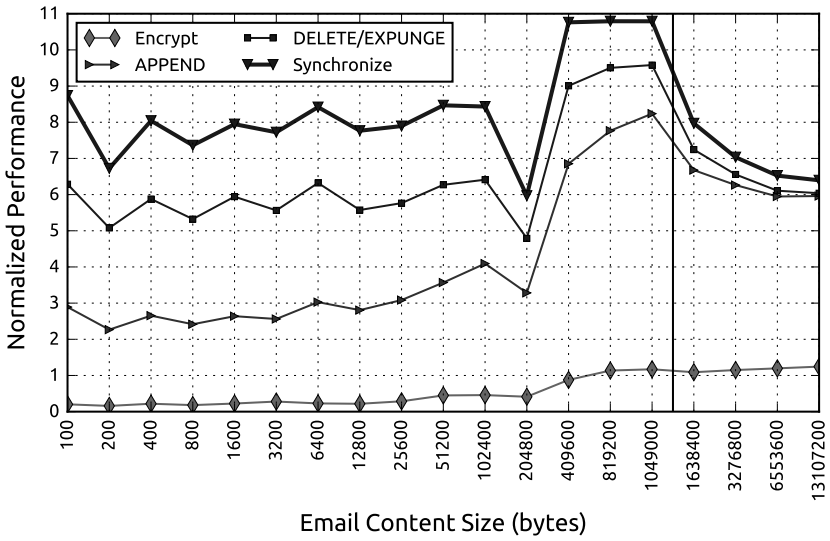

Figure 3.5: Normalized time spent for the one-time "encrypt/synchronize" relative to the cleartext download (not pictured). Points right of the line are JPEG emails.

Encrypt and Synchronize

Figure 3.4 shows the time it takes to encrypt email and replace it on the server, and synchronize the client and server. The plot labeled "Total Encrypt-Sync" includes: Encryption, APPEND, DELETE and EXPUNGE, and Synchronize. Figure 3.4 also shows the time to initially synchronize and download the original cleartext email. This is strictly not part of E3, but provides a basis to show the relative cost of E3 compared to a standard client. The time to download the cleartext email was the same for both E3 and unmodified K-9.

Before discussing the results, we highlight two important points. First, the overhead of encrypt/synchronize is a one-time cost. Once a message is encrypted and uploaded, it does not need to be processed again. Second, operations run in the background so the user is unaffected.

Figure 3.4 depicts the encrypt/synchronize time in seconds for each email size. Although the encrypt/sync time is 6× to 11× the time to synchronize cleartext emails, the overhead is not visible to users as it is processed in background threads.

Figure 3.5 shows the same encrypt/sync measurements as Figure 3.4, but normalized to the

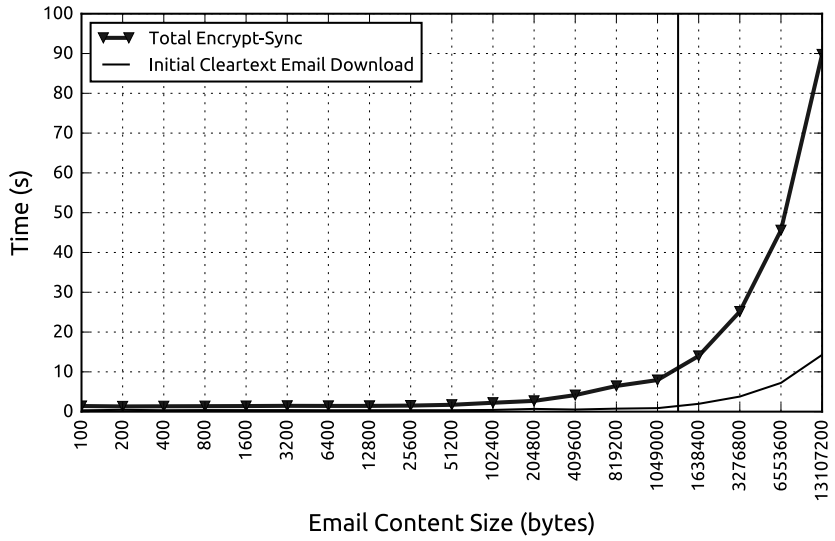

Figure 3.6: Expected time for the one-time "encrypt/synchronize" with REPLACE compared to the cleartext download. Points right of the line are JPEG emails.

cost of downloading the original cleartext email. This shows a breakdown of the relative cost of each part labeled: Encrypt, which encrypts the message; APPEND, which uploads the encrypted message; DELETE/EXPUNGE, which deletes and expunges the cleartext message from the server; and Synchronize, which verifies client-server consistency. The components are stacked so that each line is cumulative and the area between lines is the overhead for the component. For example, the total normalized overhead for 1600 B emails is 8× the initial cleartext email download, comprising of Synchronize (25%), DELETE/EXPUNGE (40%), APPEND (30%), and Encrypt (5%).

Encrypting is brief and generally takes no more time than downloading cleartext email. The cost is constant for emails smaller than 102,400 B, then grows linearly in proportion to size. This suggests that for small emails, encryption is dominated by initialization which includes generating the IV and encrypting the AES key. Once size grows beyond a critical mass, encryption time increases as well.

For small emails, the primary overhead is DELETE/EXPUNGE's multiple RTTs which are significant relative to a short APPEND time. To mitigate this overhead, clients can issue a single DELETE and EXPUNGE for batches of emails. For larger emails, APPEND (upload) dominates

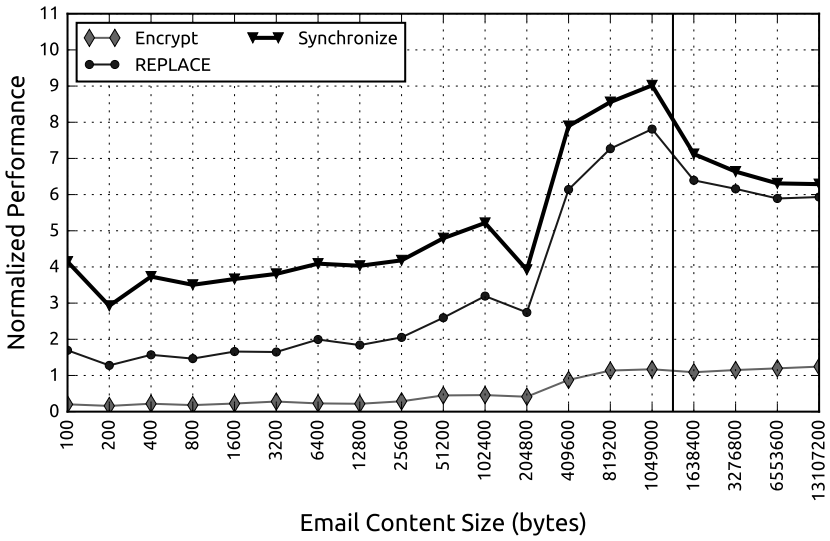

Figure 3.7: Normalized expected time for "encrypt/synchronize" with REPLACE relative to the cleartext download (not pictured). Points right of the line are JPEG emails.

for two reasons. First, uploading to Gmail was slower than downloading which magnifies the APPEND overhead. Second, the Gmail server supports Deflate/Gzip compression, and the cleartext compresses well. In contrast, ciphertexts are indistinguishable from random bits so they cannot be compressed. Thus, E3 APPENDs the full message size. However, the effects are lost for content that is incompressible. This is the case for the emails larger than 1 MB since they contained a single JPEG (incompressible) image; they consequently exhibit less overhead compared to the text emails.

The remaining overhead is due to Synchronize, which appears substantial for small messages. This involves verifying client-server consistency, updating the UI to show progress, and processing any pending commands. This constant overhead—less than a quarter of a second—is magnified for smaller emails, but becomes negligible for larger ones.

IMAP currently does not support replacing a message in a single operation. The proposed IMAP REPLACE extension [47] would eliminate the DELETE/EXPUNGE, so REPLACE's overhead will resemble APPEND alone. We approximate this by taking Figure 3.4 and removing DELETE/EXPUNGE. This leaves Encrypt and APPEND as visible in Figure 3.6. Normalized

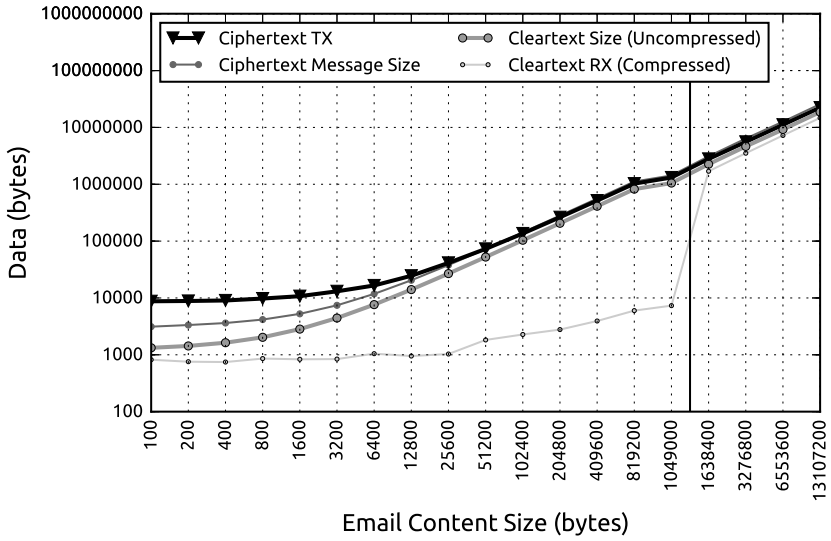

Figure 3.8: The cleartext download and a breakdown of data TX during the ciphertext upload. The y-scale is logarithmic. Points right of the line are JPEG emails.

performance can be seen in Figure 3.7. Like Figure 3.5, Figure 3.7 is stacked so that each line is cumulative and the area between lines is the overhead for the component. The reduction in the time for the worst case—small emails—is almost half.

Figure 3.8 shows the data transmitted and received to both download the original cleartext emails, and to upload and synchronize the ciphertexts. Figure 3.9 shows the same data normalized to Cleartext Size (Uncompressed), which underscores Deflate/Gzip compression's effects on the original cleartext emails. Unlike the previous normalized graphs, this figure is not stacked and therefore the plots are not cumulative. It shows both the actual data transferred when compressed as well as the true uncompressed size. Compression greatly reduces the plaintext emails, but has less benefit for the JPEG emails. Also, S/MIME incurs about a 33% overhead due to extra base64 encoding. Putting together the encoding and lack of compression, the overhead is about 10× for smaller emails. For the larger emails with incompressible JPEGs, the overhead is smaller, ranging from 64.6% to 54.1%. Unfortunately, mail clients cannot separately and lazily download attachments in encrypted email. While the extra data is high, it is an unavoidable aspect of using encryption; existing encrypted email already makes this sacrifice. Furthermore, the data

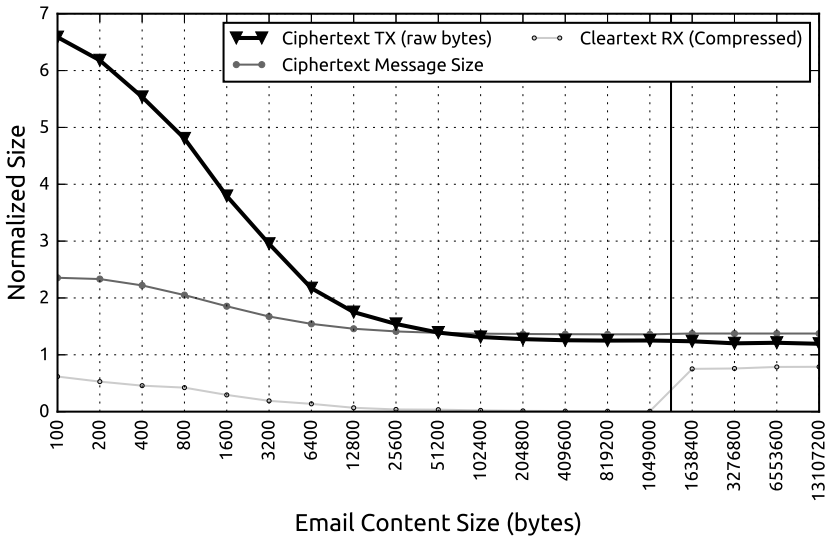

Figure 3.9: Normalized ciphertext and actual on-the-wire ciphertext transmitted relative to the uncompressed cleartext. Points right of the line are JPEG emails.

requirements of email are quite modest compared to other uses of data on mobile devices, so the difference does not significantly impact overall mobile data usage.

Decrypt and Read

Figure 3.10 shows the time-to-display (TTD) from when the user selects an email to read until the CPU completes the instructions to render the email. The TTD measures the worst-case where a client stores ciphertexts locally and must decrypt before reading. Figure 3.10 compares reading encrypted emails versus the baseline of reading cleartext emails. 100 B to 1 MB emails were two-part MIME messages with a `plain/text` and `html/text` part. Larger emails were two-part MIME messages with a one byte `plain/text` part and an attached JPEG file. Figure 3.11 shows the normalized TTD relative to the Cleartext Message TTD; the plots are not stacked and are not cumulative, like Figure 3.9.

Both Figures 3.10 and 3.11 show that the overheads are small, at 2% or less for small emails and 10% or less for larger text-based emails. The absolute overheads are generally under 20 ms, making it barely perceptible. There is a sudden drop in TTD for 1 MB or greater cleartext emails,

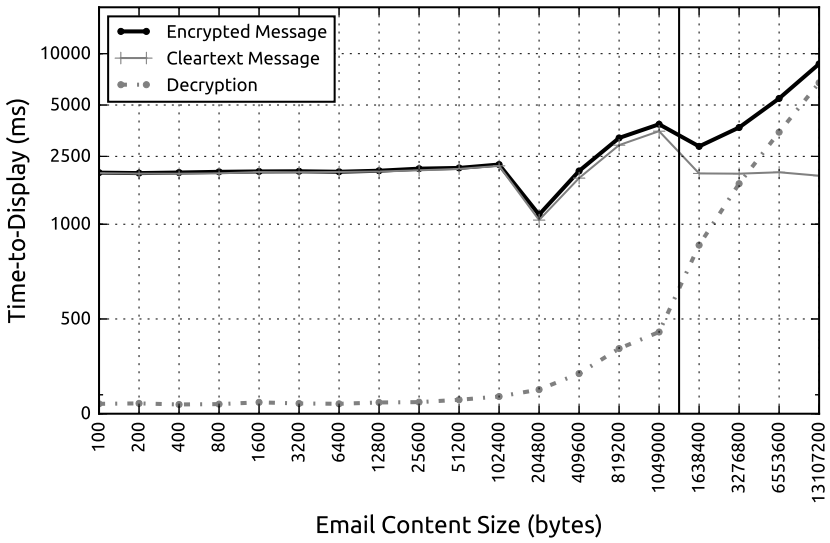

Figure 3.10: The time-to-display (TTD) for cleartext messages v. encrypted in the "worst case" (locally stored ciphertexts). Points right of the line are JPEG emails.

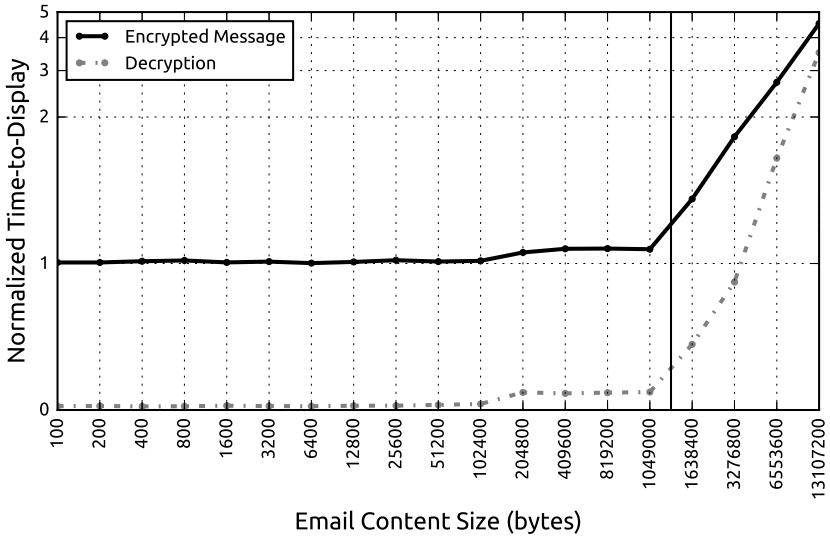

Figure 3.11: Normalized TTD for encrypted messages and their decryption time relative to the cleartext TTD in the "worst case". Points right of the line are JPEG emails.

resulting in larger overhead values ranging from 44.1% to 352.5% (4.52 in Figure 3.11). This is because the K-9 client lazily loads the JPEG image attachment only when it is actually selected. When a user reads the cleartext email, only a 1 B plaintext portion is loaded, not the attachment. In contrast, E3 decrypts the entire encrypted email, including the large attachment, resulting in higher overhead when loading the message. But when a user selects the attachment, E3 will not impose any overhead on TTD because the entire message has already been decrypted.

The overhead of decrypting is an up-front cost due to S/MIME's encryption format. S/MIME's Enveloped-Data format requires encrypting the entire body of a MIME message in a single blob, and thus no part can be read unless the entire blob is decrypted. Therefore, it is not possible to defer decryption of attachments—the entire message must be decrypted. While this is unfortunate for performance, this is not a characteristic specific to E3. This up-front cost also exists in other S/MIME mail clients, such as Apple's Mail client. A performance optimization to reduce user-perceived latency would be to proactively decrypt visible listed emails.

3.7.3 Usability

After its initial configuration, E3 by default works transparently to the user. The user thus does nothing different from using a regular mail client. As a result, E3's usability is the same as a regular mail client for everyday email usage. The main difference with E3 versus a regular mail client involves the initial setup of E3—the configuration of PDK—before a user can start sending and receiving emails. We therefore focus on the usability of the mail client setup.

We administered an IRB-approved[2] user study (protocol number AAAQ6201) with nine participants who used and compared our E3 K-9 PGP client versus an unmodified K-9 client with and without PGP. Participants used three devices we provided to them in each session: a Nexus 7 Android tablet, a Huawei Honor 5X, and a Samsung Galaxy S7. Each user was also supplied with an empty Gmail account. All participants had some experience with mobile device mail clients. They consisted of six non-technical users aged 31 to 60, and two technical users aged 21 to 30.

[2]The Institutional Review Board (IRB) is the United States' approach to an ethics committee that oversees human subjects testing.

The non-technical users all worked in blue-collar occupations or were self-employed. The technical users worked or had worked in technology, and one was a Ph.D student who specializes in computer security and mobile computing. Both had never used PGP but were familiar with its design.

The participants volunteered in 60 minute sessions in which they role-played as a tax accountant using email to request a client's tax forms. The 60 minute session comprised three 20 minute sessions, each devoted to using vanilla K-9, E3 K-9 PGP, or K-9 with PGP. During each 20 minute session, we instructed the user to configure the selected mail client with a Gmail account then send and receive emails to obtain tax forms from three separate people. More specifically, the user first set up the respective email client with an empty Gmail account on one of the three mobile devices, requested a tax form from the first person, and verified the response was encrypted (for E3 and PGP) by checking for the visual encryption flag indicator on the K-9 client. Upon successfully completing the first email exchange, users then configured a second device with the same Gmail account, which essentially tested E3 and PGP's key management. E3 required completing the two-way verification to distribute E3's public keys, and PGP required transferring their single private key. If successful, users then requested the tax form from the second person. This was then repeated for the third device and person.

We provided users with a visual setup guide for both E3 and PGP, and they could ask the study coordinator for help with specific errors (for example, if they unknowingly made a typo or didn't know how to go to the home screen), but we provided no in-depth help. Our reasoning was to strike a balance between providing consistent help to all users for both solutions while also preventing cases where users would get stuck on a simple mistake unrelated to the study goals. To mitigate the effects of short-term memory on survey results, we randomized the order of the email clients. To avoid priming participants for favorable responses, we explained our research purpose only after the surveys had been completed.

After participants completed their tasks or reached the 20 minute limit per client, they completed the System Usability Scale (SUS) [61], an industry-standard questionnaire also used in many

Soln.	Count	Mean	Std. Dev	Min.	Q1	Median	Q3	Max
K-9	8	81	15	63	70	76	96	100
E3	8	74	19	48	60	76	91	98
PGP	8	41	12	28	28	45	50	58

Table 3.3: System Usability Scale summarized scores.

similar studies [18, 62, 23, 22], for the system they had just used. At the end of the study, participants completed 14 additional survey questions specific to our research, and a final free-form question requesting any comments. To ensure that participants actually understood each email solution they used, the study coordinator explained each system prior to completing the 14 additional survey questions.

The summarized SUS scores are presented in Table 3.3. A higher score means better usability. The results for K-9 and E3 were quite close while K-9 PGP received remarkably low ratings. This suggests that users felt that E3 was almost as easy to use as K-9, while PGP was significantly worse. All users except the one technical user who specializes in computer security and mobile computing failed to complete K-9 PGP's tasks in the time limit even with copious help. The pain point in PGP where users struggled was the private key management when they had to transfer their PGP keypair to their other devices. On the other hand, all users succeeded in every instructed step for E3 K-9 PGP in 10 to 15 minutes. The average completion time for K-9 was 8 minutes.

We also asked users to compare the email solutions which we summarize in Table 3.4. Responses are on a scale of 1 (Strongly Disagree) to 5 (Strongly Agree). Subjects in general agreed that E3 was easier to use than PGP, but E3 still introduced noticeable extra setup time compared to K-9. For many of these questions, users choose a score of 3 despite giving similar usability scores for K-9 and E3 as seen in Table 3.3. This suggests that another factor unrelated to usability influenced their responses to our custom questions. The most likely culprit is that most of the non-technical users did not consider themselves important enough to use encryption. They thus tended to be indifferent and responded with the middle-ground score of 3 for any question concerning the encrypted email solutions.

Free responses included saying "PGP sucks" and "I had no idea what I was doing with [PGP]." Several subjects commented on how much easier E3's two-way verification was compared to PGP's key exchange and private key import/export. Most users felt that they were not important enough, i.e.not public figures or celebrities, to be targeted by attackers and therefore did not need to use encryption, but could still see the value in having an easier to use email encryption solution for people who do handle sensitive data.

It is important to note that our user study places a large emphasis on configuring email clients, which is a relatively infrequent occurrence. Furthermore, it is not uncommon for non-technical users to ask others, technical support in the context of an enterprise organization or customer support when purchasing a device, for assistance in setting up a device. The fact that the main usability difference between E3 and vanilla email is in the client configuration and that there is no difference for sending and receiving email suggests that E3 usability is likely to be even better in practice.

We draw these conclusions: (1) E3 is easy to use even for non-technical users. (2) E3 is much more usable and intuitive than PGP. (3) PGP is too unwieldy to actually be used. Overall, our user study results were very positive in favor of E3, but further studies with more users and a wider range of activities would be illuminating.

3.8 Related Work

The seminal "Why Johnny Can't Encrypt" paper illuminated the confusing process of encrypting email and showed how inaccessible PGP is to average users [13]. They found that correctly sending encrypted email in an end-to-end encrypted email setting is outstandingly difficult.

Many works following "Johnny" have tried to tackle the problem of end-to-end encrypted email by attempting to make the process easier or more transparent. STREAM [63] uses SMTP/POP proxies which opportunistically encrypt email by finding keys or generating them on the fly, but key management is problematic. Verifying keys involves out-of-band communication such as phone calls, and delivering keys requires users to know how to extract keys from email, install them, and

use them. STEED [64] extends the IMAP standard to support transparent end-to-end encryption, but requires modified clients and servers, and does not address key management at all. Pwm [18] and Pwm 2.0 [62] attempt to make end-to-end encryption transparent by integrating with popular mail providers and relying on a third-party identity-based encryption (IBE) server to manage keys, but users authenticate to the IBE server via their email account which does not protect against compromised accounts. Confidante [65] leverages users' ubiquitous use of social media accounts to ease public key management and verification via a third-party service. It however relies on users being able to correctly identify social media accounts.

Various commercial services that provide end-to-end encryption try to address the key management directly by taking a walled garden approach. Lavabit [66], Posteo [67], and Tutanota [68] create closed platforms where the service handles all key management on its servers, but users are restricted to encrypting messages only to other users of the same platform or to redirecting recipients to the service's website to gain access to an encrypted file. Services such as Lavabit maintain master keys which could decrypt all emails, making them vulnerable to compromises and subpoenas.

Autocrypt [54] is a decentralized and incrementally deployable system for distributing public keys to support end-to-end encryption by making public key management more usable. Only clients need to be modified to support Autocrypt. Autocrypt includes the sender's public key in an email and an indicator whether the sender prefers encryption. The receiver replies in the same manner, including his public key in the email and an indicator whether encryption is preferred. Autocrypt thereafter will send encrypted email between the two parties if any of three criteria are satisfied: the sender requests encryption, the received email was encrypted, or all parties explicitly prefer encryption. Autocrypt does not encrypt all of a user's email, for example an email from someone who does not prefer encryption. Given that Autocrypt use remains limited, it may not protect a substantial portion of a user's emails if a compromise occurs. This is in contrast to E3, which will protect all of user's email before a compromise occurs. E3 could be used to complement Autocrypt, most obviously by encrypting plaintext emails with non-Autocrypt correspondents.

Another critical difference is that Autocrypt eases PGP public key distribution but does not address private key management and has no solution for making it easy to read encrypted email on multiple devices.

E3's encrypt on receipt approach has been proposed using other mechanisms. Most examples modify one's Mail Transfer Agent or Mail Delivery Agent to encrypt emails before delivering them to the client [69, 70], but this is too complicated for non-technical users. Posteo [71] provides support for encrypting emails on receipt, but their approach is server-side and only works on their servers. Unlike E3, none of these approaches work with existing unmodified IMAP servers and clients, and none of them address the issue of client-side key management.

3.9 Summary

Easy Email Encryption (E3) introduces a new client-side encrypt-on-receipt mechanism coupled with a concrete design and implementation of our Per-Device Keys (PDK) key management system, both of which are compatible with the existing IMAP standard and servers. E3 email clients automatically encrypt received email without user intervention, making it easy for users to protect the confidentiality of all emails received prior to any email account or server compromise. E3 uses keys that are self-generated (and self-signed for certificates), and PDK makes it easy to use them to access encrypted email across multiple devices. Users no longer need to understand or rely on public key infrastructure, coordinate with recipients, or figure out how to use PGP or S/MIME. We show that E3 is easy to implement on a variety of platforms including Android, Windows, Linux, and even Google Chrome, and show that it works with popular IMAP-based email services including Gmail, Yahoo! Mail, AOL, and Yandex Mail. Our user study results show that real users, even non-technical ones, consider E3 easy to use even when compared to using regular unecrypted email clients and vastly easier to use over the state of the art for PGP. Our measurements using E3 with Gmail services show that performance overheads are modest and acceptable in practice.

Almost exactly 20 years ago, Johnny was unable to encrypt. In the current modern era, the explosive growth of ubiquitous and always-on, always-connected mobile devices has provided the

necessary foundation for putting a new and usable spin on the idea of receiver-controlled encryption. Johnny could not encrypt in his time, but Joanie in the modern age certainly can.

#	Question (1 = Strongly Disagree, 5 = Strongly Agree)	Mean	Std.	Min.	Med.	Max
31	I found it easy to use K-9.	4.50	0.55	4	4.5	5
32	I found it easy to use K-9 with PGP.	2.17	0.75	1	2	3
33	I found it easy to use K-9 with E3.	3.83	0.98	3	3.5	5
34	I could see myself using K-9 on a regular basis.	4.00	0.89	3	4	5
35	I could see myself using E3 on a regular basis.	3.83	0.75	3	4	5
36	I could see myself using PGP on a regular basis.	2.00	0.89	1	2	3
37	I thought E3 takes too long to set up each time on a new device.	2.00	0.89	1	2	3
38	I thought PGP takes too long to set up each time on a new device.	3.67	1.21	2	3.5	5
39	I thought that E3 was easier to use than PGP.	4.17	0.98	3	4.5	5
40	I thought that using the QR code scanner was harder than verifying a three word phrase.	3.50	1.22	2	4	5
41	I thought that transferring my key in PGP was harder than verifying a three word phrase in E3	4.17	0.98	3	4.5	5
42	The extra security with E3 is worth the extra steps compared to regular email.	4.17	0.98	3	4.5	5
43	The extra security with PGP is worth the extra steps compared to regular email.	2.50	0.84	1	3	3
44	The extra security with PGP is worth the extra steps compared to E3.	2.00	0.63	1	2	3

Table 3.4: Summarized scores for added survey questions. (Questions are abbreviated for spacing reasons.)

Chapter 4: Easy Secure Photos (ESP)

4.1 Introduction

The rapid proliferation of smartphones with increasingly high-quality built-in cameras is driving enormous growth in the number of photos being taken, with well over a trillion photos captured each year [72]. Since smartphones often have low storage capacity and are prone to accidental damage and loss, many users use the cloud to permanently store their photos online via cloud photo services such as those offered by Google, Apple, Flickr, and others. Google Photos is particularly popular, given its promise of unlimited storage capacity at no charge until June 2021, with over a billion users [73]. However, users' photo collections often represent a gold mine of personal information which is valuable not only to the services, but to attackers as well. Even if users trust cloud photo services with their data, the threat of attackers compromising user accounts and data is tangible. External attackers often target one of the weakest points of account security, passwords, to gain access to personal photos such as in the case of the 2014 celebrity nude photo hacks [74]. Because passwords are such a weak defense that is often compromised through social engineering, phishing, or password leaks, many services augment account security with two-factor authentication (2FA), but this is still not enough. One important reason is because adversaries may also be internal, such as rogue employees at cloud services abusing their access privileges to snoop on user data [6, 7, 8, 9], and bugs or errors may reveal user data to unintended recipients, such as the recent case of Google Photos accidentally sharing users' private videos with other completely unrelated users [75]. Regardless of attackers' origins, it takes only a single compromise of a user's account to expose their entire photo collection.

Encryption offers a well-known solution to this problem: users can simply encrypt all of their photos before uploading them. Then even if an attacker compromises user accounts, such as by

phishing account passwords, their photos are indecipherable. The problem is that existing encryption schemes are incompatible with cloud photo services. Google Photos expects uploaded files to be valid images, and compresses them to reduce file sizes. Image compression is incompatible with general and photo-specific encryption techniques, causing corruption of encrypted images. Even if image compression were compatible, mobile users expect to be able to quickly browse through thumbnails of their online photo collections, which are typically generated by cloud photo services who need access to photo data to generate meaningful thumbnails; this is not possible with any existing photo encryption schemes. Finally, encrypting data and managing keys is too complicated for most users especially if public key cryptography is involved [13]. This is made more difficult for modern users who commonly access their photos from multiple mobile devices which each must decrypt their photos. While some third-party photo services promise image encryption and user privacy, and others propose new external secure photo hosting services [76, 77, 78, 79, 80, 81, 82], they require users to abandon existing widely-used cloud photo services such as Google Photos and their desirable features, including free and unlimited photo storage.

To address this problem, we have created Easy Secure Photos (ESP), a system that enables mobile users to use popular cloud photo services such as Google Photos while protecting their photos against account compromises. ESP encrypts uploaded photos so that any attackers that compromise user accounts only have access to encrypted photo content, yet the encryption is transparent to authorized users who can visually browse and display images in largely the same manner as when using the cloud photo services with unencrypted images. ESP achieves this functionality by introducing a new client-side encryption architecture that is compatible with and requires no changes to cloud photo services such as Google Photos, and has no reliance on any external third-party system or service provider. The architecture includes three key components: a format-preserving image encryption algorithm, an encrypted thumbnail display mechanism, and a PDK-based easy-to-use key management system.

ESP provides an image encryption algorithm which is compatible with lossy and lossless image formats such as JPEG and PNG, and with image compression techniques used by many cloud

photo services; and is also efficient enough for mobile devices. Our algorithm converts an image to RGB, encrypts it in the RGB color space using a block-based Fisher-Yates shuffle [83], splits the RGB channels into three separate grayscale ciphertext images, then converts them back into the original image format, including compression as needed for image formats such as JPEG. The novel encrypted grayscale approach maintains the original image dimensions and ensures compatibility with standard JPEG compression, newer widely used compression techniques such as Guetzli [84, 85] JPEG encoding, and JPEG chroma subsampling. The encryption also uses a per-image encryption key.

ESP also implements an encrypted thumbnail display mechanism that is simple and easily compatible with cloud photo services. ESP simply uploads its own encrypted thumbnail to the cloud and redirects the client-side image browser to view the uploaded client-generated thumbnails rather than any server-generated thumbnails. This approach makes it easy for users to interactively browse and view thumbnails on any device authorized to access the encrypted images.

ESP also includes an instantiation of PDK, modified to support the case of photo hosting services in contrast to E3 as described in Chapter 3, to provide an easy-to-use key management scheme that allows users to access encrypted photos from multiple devices while eliminating the need for users to know about and move private keys from one device to another. Like E3, ESP's version of PDK uses self-generated keypairs and a verification process that builds up a chain of trust from one device to another, and benefits from using multiple devices such as smartphones and computers. However, since the context of photos differs from that of email, ESP instead uses the cloud photo service itself as a communication channel for a QR code-based message protocol.

We have implemented ESP on Android in the Simple Gallery app, a popular photo gallery app with millions of users, adding the ability to use Google Photos for unlimited encrypted photo storage. ESP satisfies the system design presented in this dissertation, consisting of only client-side modifications that use the Google Photos API and requiring no changes to the Google Photos cloud service, while providing key management with an instantiation of the PDK concept. We have evaluated ESP using images from Google's Open Images Dataset V5 [86]. Our experimental results

show that ESP (1) works seamlessly with Google Photos even with their image compression techniques, (2) produces encrypted images with quality comparable to the original images when both are stored in and processed by Google Photos, (3) provides strong security against practical threats including account compromises and analysis by machine learning image classifiers, (4) provides fast encrypted image upload, download, and browsing times with modest overhead when using Google Photos, and (5) has a usable key management scheme. ESP is easily adaptable to any other photo site that has the concept of albums, though it would be necessary to verify compatibility with its compression algorithms.

The contributions of this work are the design, implementation, and evaluation of a new system for encrypting images stored on existing photo hosting services, with no server-side modifications, requiring no trust in the photo services or their servers. The system includes a novel format-preserving image encryption scheme coupled with a key management solution for users. To the best of our knowledge, we are the first to address practical issues such as key management, usability, image sharing, and compatibility with existing photo hosting services without needing to trust them, all in a single system.

4.2 Threat Model

ESP's purpose is to protect the privacy of images stored remotely in cloud services with no changes to software or protocols other than client-side installation of a ESP app. The cloud service provider may be malicious or an attacker may compromise user accounts by obtaining passwords, or by bypassing password checks entirely through abusing privileged access. They may be sophisticated but not at the level of a nation-state intelligence agency.

We assume that the devices with ESP clients that encrypt and upload images to cloud photo services are secure and trustworthy. We do not attempt to protect users' devices because this is an orthogonal concern that should be managed by device hardware or at the operating system level. A compromise of a user's device would mean the attacker has access to the private keys that can be used to decrypt any encrypted images belonging to the user.

4.3 Usage Model

ESP is easy to use. A user simply installs an ESP photos app, then authenticates it with a cloud photo service such as Google Photos. We have built an Android ESP app for use with Google Photos, and the only setup step is a user selecting his device's Google account. The app appears no different from a regular photos app, except that it encrypts images before uploading them to Google Photos and decrypts them on download. The encryption and decryption are transparent to the user. Users are free to perform common operations: viewing thumbnail galleries, moving photos to albums, assigning labels, modifying metadata, editing pictures, and sharing them with others.

ESP assumes that users only access encrypted images on trusted devices which is reasonable as it is uncommon (and inadvisable) to view photos on untrusted devices such as public computers. In other words, ESP is not compatible with using untrusted computers such as those at an Internet cafe, nor should it be if users care about their privacy.

Any device that a user trusts can decrypt and view image content from the cloud photo service. Users are free to use ESP on as many devices as desired. When a user configures multiple devices, each app installation on a new device after the first requires a short and simple setup step to synchronize it with any previously configured devices via a platform-independent verification process. This process appears conceptually similar to device pairing: a user verifies his new ESP device using one of his existing ESP devices. However, in contrast to normal pairing, the user only needs to complete this synchronization step *once* per new device with only one other existing ESP device for all his other ones to recognize it.

The verification consists of these steps: (1) the user configures a new device with ESP, (2) it displays a random phrase, and (3) the user copies the random phrase to *one* other existing ESP device. Successfully completing these steps ends the new device's setup, allowing the user to use it to upload and download encrypted images from his chosen cloud photo service.

Users can remove devices from their ESP ecosystem if they wish to revoke access to their

photos. Revocation can be completed on any configured ESP device to remove any other device. The user may also choose the reason for removal which will affect his devices' next steps. If the user indicates that he is removing the device because it was lost, ESP informs that user that his photos will be re-encrypted for security reasons.

ESP's design intrinsically helps protect users from losing access to their encrypted photos assuming they have more than one ESP device. A user may lose one of his devices, but can still access his encrypted photos on the remaining ones. However, if a user loses all of his devices, then the user may provide a recovery password to regain access to his encrypted photos; this is a standard practice on popular operating systems for disk encryption schemes [87, 26].

Since ESP apps encrypt the images stored in the cloud photo provider, apps which do not support ESP will be unable to decrypt them. The encrypted images, however, retain their original file formats and therefore can be opened in image apps and editors for viewing the ciphertext, and are accepted as valid JPEG files when uploading to hosting services. Users of services with web browser interfaces may install ESP web browser extensions to decrypt and view their photos.

4.4 Architecture

In addition to the general system design described in Chapter 2, ESP is designed for compatibility with existing cloud photo services, preservation of image formats when encrypted, and end-user usability with respect to browsing images and managing keys. While ESP works with multiple cloud photo services, we focus on Google Photos given its popularity. Google has no stipulations in its terms of service that prohibit users from encrypting their data [10].

4.4.1 Format-Preserving Encrypted Images

ESP's image encryption algorithm is compatible with cloud photo services which support standard image formats including JPEG, PNG, WebP, and RAW. Compatibility means uploading encrypted images without them being rejected, and decrypting them with minimal loss of quality beyond any compression by the service. We focus on JPEG since it is the most commonly used

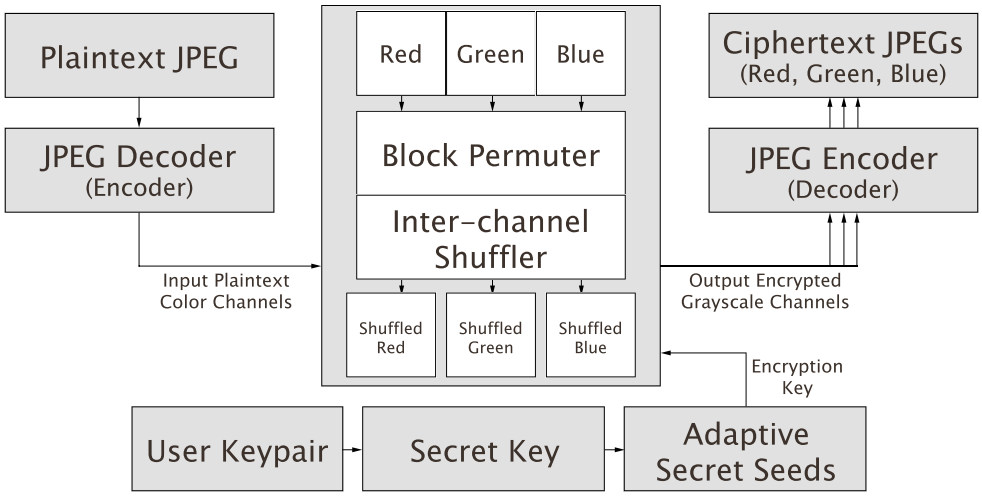

Figure 4.1: The ESP encryption architecture for JPEGs.

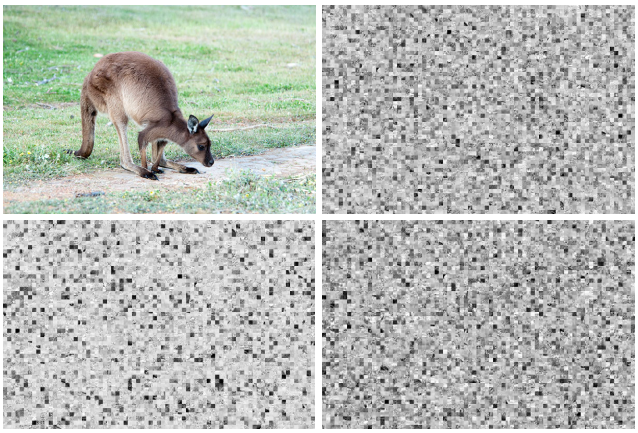

Figure 4.2: An image and its encrypted RGB components.

image format, though ESP's encryption method works with others. We also focus on Google Photos' free tier of service which compresses and processes uploaded images. Google Photos does not publicly specify its processing pipeline, but we observe it to downsample the JPEG chroma format to 4:2:0, and to apply compression with possibly a noise filter. The compression is likely a standard JPEG quantization plus Google's own Guetzli JPEG encoder [84, 85]. ESP encryption must account for these techniques or images will be corrupted. It must also be fast and efficient, as many users use resource-constrained mobile devices.

To understand how ESP's encryption works and the problems it addresses, it is necessary to understand JPEG compression. Images rendered on a user's screen generally consist of pixels,

each of which has red (R), green (G), and blue (B) components, and each component is an 8 bit value in the range 0 to 255. Converting RGB data into JPEG format takes four steps. (1) The RGB components are converted into luminance, chroma-blue, and chroma-red (YCbCr) components. Sometimes the chroma is subsampled to reduce the sizes of the Cb and Cr components which are less important for image quality. For example, an image with full size, half size, and quarter size CbCr components are in 4:4:4, 4:2:2, and 4:2:0 format, respectively. (2) YCbCr components are transformed using a discrete cosine transform (DCT) which outputs DCT coefficients. (3) The DCT coefficients are quantized, thereby reducing the number of bits needed per coefficient; quantization is the main lossy compression step of JPEG and is controlled via the JPEG quality parameter. Quantization is performed on 8×8 blocks of pixels. (4) Finally, lossless compression techniques are used to further reduce the image file size. Cloud photo services' compression practices are problematic for encrypted images due to the data loss caused by quantization and chroma subsampling. We have confirmed images encrypted using common DCT coefficient diffusion and confusion techniques experience significant visual corruption when compressed and are unusable.

ESP is an encrypt-then-compress design that shuffles 8×8 pixel blocks as shown in Figure 4.1. Figure 4.2 shows a sample image encryption produced by ESP. ESP's encryption is robust against JPEG compression because it does not modify the values or positions of pixels within the 8×8 blocks, so any pixel-based lossy operations are orthogonal to ESP's decryption. Consider a simple image with only one grayscale component divided into 8×8 blocks which is encrypted by shuffling the blocks. Decryption therefore means moving each block back to its original position, so any lossy compression of the pixel data in each block is unrelated to the decryption. In contrast, standard image encryption methods modify pixel values and also shuffle pixels within each block. Lossy compression also modifies these encrypted pixel values, but in a non-uniform manner, making it impossible to reconstruct the original values upon decryption. This resulting corruption appears visually, and is even worse for color images.

As shown in Figure 4.1, encryption and decryption are performed on the RGB color data, so encryption occurs after the image has been decoded and decompressed to the RGB color space.

Encryption outputs three separate grayscale ciphertext JPEG images created from encoding and compressing the encrypted RGB channels, each representing one of the RGB color channels. Likewise, ESP decrypts JPEGs after the three ciphertext JPEGs are passed to the JPEG decoder and converted to RGB values. Each grayscale ciphertext is decompressed and converted to RGB, then the resulting plaintext decrypted RGB data is rendered for viewing or compressed again to be stored on disk. Since ESP JPEGs are legitimate JPEGs, they can be decoded and displayed like regular JPEGs, but the decoded results do not reveal the original images. The use of grayscale ciphertext images makes ESP also immune to chroma subsampling because the images have no chroma components, only a luminance (Y) component. One beneficial side effect of this is that ESP images can retain higher resolution chroma channels in the decrypted JPEGs compared to unencrypted images, which may have their chroma subsampled by Google Photos.

A popular approach to format-preserving encryption is to encrypt the data via a pseudo-random number generator (PRNG) with known secret seed values. ESP applies this approach in two main components: block-based pixel permuting, and inter-channel shuffling of these permuted blocks. The identifier for the actual algorithm used to encrypt an image is stored together with the image in its EXIF data, similar to how widely used cryptosystems such as TLS support multiple algorithms. Like these systems, ESP allows the addition of new crypto standards. Here we define an example of such an encryption method which we name ESP-FY. To begin encrypting an image using ESP-FY, ESP computes three initial secret seed values, (s_R, s_G, s_B), one for each RGB component, to derive (s'_R, s'_G, s'_B) values with the appropriate seed length to use as the actual inputs to the PRNGs. Note that (s_R, s_G, s_B) are unique per image, making them adaptive secret seeds generated from a user's stored secret key and properties of the image. Generating them is discussed further in Section 4.4.2.

ESP first decodes a JPEG and converts it to RGB. It then pads the image's dimensions to the nearest multiple of 8—the same strategy as the JPEG format. Then, ESP-FY's first encryption process permutes the image in 8×8 pixel blocks. Figure 4.3 depicts an example of this applied to a 4×2 block, or 32×16 pixel, image. It consists of applying a Fisher-Yates shuffle to all

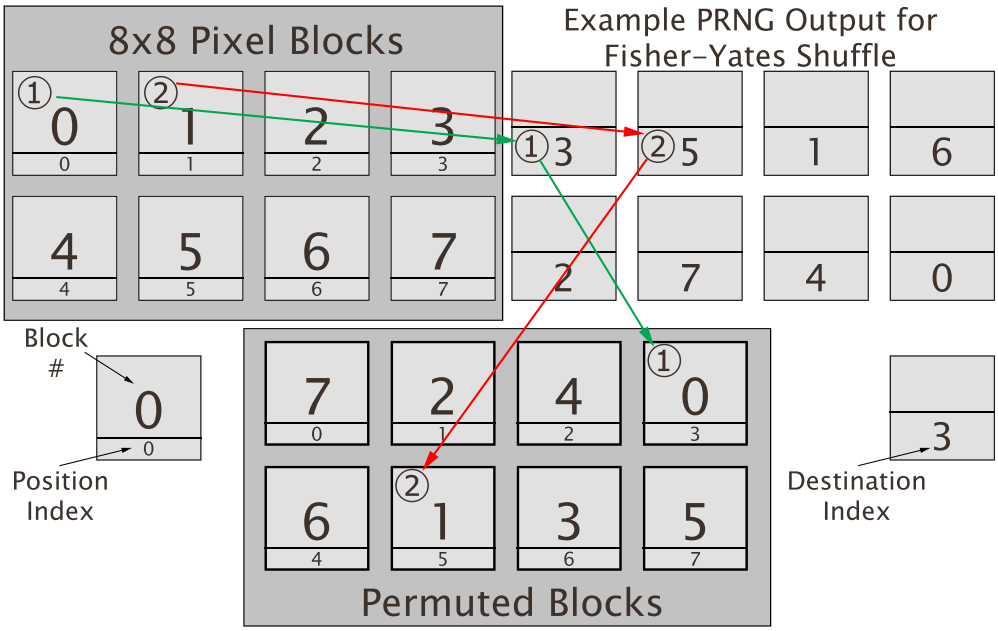

Figure 4.3: Example of 8×8 block permutation.

the 8×8 blocks in each RGB color channel. The shuffle is driven by three PRNGs, one for each RGB channel, seeded by the previously described (s'_R, s'_G, s'_B) 256 byte secret seeds. They may be truncated if they are too large for the PRNG. Once shuffled, the RGB channels are each compressed to three separate grayscale JPEG images. The user may choose a JPEG quality setting that represents a trade off between preserving the image's visual quality, and larger ciphertext file sizes.

The second part of encryption is an inter-channel shuffle of blocks across the RGB components. ESP pseudo-randomly swaps the ith block in each iteration of the Fisher-Yates shuffle with the ith block of the other channels. In other words if we are in the ith iteration of the Fisher-Yates shuffle, then blocks (i_R, i_G, i_B) in each of the RGB channels are shuffled. Which swaps occur is determined by $s'_{RGB} = s'_R \oplus s'_G \oplus s'_B$ as the input to a PRNG. Then on each block iteration, generate a uniformly distributed random integer $k_i \in [0, 5]$. Each k_i represents a unique ordering of (i_R, i_G, i_B) of which there are $3! = 6$ permutations. Assume that $k_i = 0$ is mapped to the ordering (i_R, i_G, i_B) with no swaps. Now consider $k_i = 1$ mapped to (i_R, i_B, i_G) which means that the block i_G is swapped with block i_B, i.e. the ith green block is swapped with the ith blue block. Therefore, each $k_i \in [0, 5]$

generated by each round of shuffling is mapped to a unique ordering of (i_R, i_G, i_B).

The encryption algorithm is therefore a Fisher-Yates block shuffle in each RGB channel, which on each iteration pseudo-randomly swaps blocks (i_R, i_G, i_B) across channels. Decryption is the inverse in reverse order: reversing the inter-channel shuffle on each iteration over the images' blocks while reversing the Fisher-Yates shuffle of the 8×8 blocks of pixels.

The security of shuffle-based encryption scales with the image dimensions. Smaller images have fewer permutations, and larger images have more. Block-based shuffling is not ideal for very simple images, such as ones where the visual data is aligned to the blocks, e.g., solid colors or simple patterns. However from a privacy perspective, simple images are less likely to contain sensitive information. Finally, like all encryption schemes, an encrypted image which has been resized cannot be decrypted. ESP therefore resizes images larger than the maximum allowed sizes for photo hosting services prior to encryption. Google Photos' maximum size is 16 MP, so ESP scales images to at most 16 MP before encrypting them. This is no worse than a user uploading an unencrypted image larger than 16 MP normally since it will be resized in any case.

4.4.2 Adaptive Secret Seed Generation

A crucial component of encryption is the selection of the secret seed values for the PRNGs. An easily guessed seed is a security risk. Accordingly, ESP adaptively generates the seed values (s_R, s_G, s_B) to be unique per encrypted image.

A user's initial secret key S is a sufficiently long, say 256-bit, string of cryptographically secure random bits. S serves as the key to encrypt the adaptive seed values (s_R, s_G, s_B) unique to each image. Each of these adaptive seed values is 384 bits and they are generated separately for each image, then encrypted using S as $E_S(s_R, s_G, s_B)$. This is stored in the encrypted grayscale images' metadata either as EXIF format or via another method such as Google Photo's metadata fields.

4.4.3 Encrypted Thumbnails

A difficulty in supporting image encryption with existing cloud services is constructing a seamless experience with fast response times so that users can quickly browse through thumbnail galleries of their images. Without any special attention to this matter, a user would only be shown thumbnails created from the ciphertexts, making them unusable for identifying images, or would have a sluggish experience due to the delay caused by downloading and decrypting the full resolution image to generate decrypted thumbnails. A simple solution would be for ESP clients to create and generate thumbnails from newly encountered encrypted images, but this is a laborious process involving all of a user's ESP clients downloading all the images at full resolution, decrypting them, resizing them, and generating thumbnails. The essential problem here is that this approach adds too much overhead, in the forms of processing time and network bandwidth, to the critical path of a user trying to browse his images for the first time on a device.

It is much more palatable for users if the overhead is primarily incurred when uploading a new image or batch of images, as a user is free to continue using the app while ESP processes the images in the background for uploading. ESP therefore prepares two encrypted images (six grayscale ciphertext images) when a user uploads an image to Google Photos. The first is the original image in encrypted form, and the second is a resized thumbnail of the original image which is separately encrypted. The encrypted original is stored in the user's chosen location on Google Photos while the encrypted thumbnail is uploaded to an album specifically for encrypted thumbnails; the ESP client hides this album from the user under normal operation.

When the uploads complete, the user's ESP client creates a mapping of the encrypted images' Google Photos media IDs with the encrypted thumbnails' media IDs. Thus, when the user is browsing an encrypted album, his client requests the smaller size encrypted thumbnail rather than the full size originals. The client requests and downloads full size encrypted images at a lower priority than the thumbnails when in gallery mode. However, it may prioritize requesting full size images that the user is likely to view next, for example if the user is swiping through individual images. The user's other ESP clients that do not have the mappings of encrypted originals to encrypted thumb-

nails are able to recreate them independently by periodically scanning the special album for new encrypted thumbnails. Google Photos provides free unlimited storage for images smaller than 16 MP—more than enough for even larger size thumbnails—so remotely stored encrypted thumbnails do not waste a user's storage quota.

4.4.4 Image Sharing

ESP users have two options for sharing their images and albums with others: normally or securely. Normally sharing images with others appears the same as sharing a regular unencrypted image. If the source unencrypted image is available locally on the user's device, it is uploaded unencrypted as a separate copy to the service and shared with the recipient via the service's normal mechanism. If the image is not available locally, the image in Google Photos is downloaded, decrypted, and re-uploaded in plaintext format to be shared with the recipient. The user is notified that normal sharing is insecure and may compromise the privacy of the image. However, the willingness to share a photo with another user already represents a security risk as no guarantee can be made that the recipient is trustworthy. For example, the recipient may make a copy of the image; the sender has no control over this.

ESP users may securely share their images and albums with other ESP users. Suppose Alice wishes to share an encrypted photo album with Bob. If this is their first time sharing albums with each other, they perform a one-time handshake which is a public key exchange. Alice may ask Bob to shares his handshake link (URL) which is generated by his ESP app. Bob's ESP public key and other metadata are encoded into this URL; this is then shortened, for example, via Google Firebase Dynamic Links, so that if Alice opens it on Android or iOS, it will be routed to the ESP app and add Bob's public key to its key chain. Alice may also share her handshake URL with Bob whose ESP app performs the same process. This completes the public key exchange between Alice and Bob without either of them needing to know what a public key is.

Now Alice begins sharing $E(Album_A)_{S_A}$ which is the encryption of the images in $Album_A$ using her secret key (S_A). Alice selects $Album_A$ to share and chooses Bob from her list of

known ESP users. Alice's client creates a new album $Album_{AB}$, invisible to Alice, to share with Bob; the original album $Album_A$ is not shared. Alice's client then generates S_{AB}, a new secret key shared by Alice and Bob. Alice decrypts $E(Album_A)_{S_A}$, and re-encrypts and stores the result in $Album_{AB}$, resulting in $E(Album_{AB})_{S_{AB}}$. Alice then encrypts S_{AB} using both her own and Bob's public keys, resulting in $E(S_{AB})_{(Alice,Bob)}$. Next, if Alice's cloud photo service supports its own native limited sharing mechanism, Alice's client ensures that $Album_{AB}$ is viewable by Bob's account and retrieves the service's URL for $URL(Album_{AB})$. Finally, Alice's client presents her with a shortened share URL $U_{Alice,Bob}$ to provide to Bob. When expanded, $U_{Alice,Bob} = \{URL(Album_{AB}), E(S_{AB})_{(Alice,Bob)}\}$. Alice copies $U_{Alice,Bob}$ and sends it to Bob whose device automatically opens it in ESP. Bob's client uses his private key to decrypt $E(S_{AB})_{(Alice,Bob)}$ and retrieve S_{AB} to decrypt the images in Alice's album.

Alice and Bob's clients also support viewing $Album_{AB}$ on multiple devices. When Alice's client constructs $U_{Alice,Bob}$, it is also synchronized among all of Alice's devices in the form of a PDK broadcast message, the protocol for which is discussed in Section 4.4.6. Any of Alice's devices which trust the broadcasting device will accept the message, decrypt the contained encrypted shared seeds, and record the seeds' association with $URL(Album_{AB})$. Bob's client performs the same steps to synchronize the shared seeds among his devices. If Alice chooses multiple individuals including Bob with whom to share $Album_A$, ESP adds their public keys to the list of keys used to encrypt the shared seeds $E(S_{AB...})(Alice, Bob, ...)$.

Alice revokes Bob's access to $Album_{AB}$ by removing him from $Album_A$. Alice's client first revokes any granted access controls via the cloud service itself for Bob if applicable, and then deletes $Album_{AB}$ from the service. ESP cannot prevent Bob from accessing any copies of $Album_{AB}$ that he may have saved. Alice's revoking Bob's access only prevents Bob from viewing the album via the cloud service. If Alice has shared the album with multiple individuals, then she may only revoke access to all of the individuals at once. If Alice desires granular access revocation, she must share her album with each user separately and provide them with individual share URLs.

4.4.5 Other Features and Limitations

ESP is incompatible with cloud photo services' features that rely on server-side access to photo image data. This includes facial recognition and detection, machine learning-based labeling and classification, image searching, and other similar functionality. For example, Google Photo uses server-side machine learning to classify images ostensibly for supporting search. However, this feature is incompatible with encrypted images, as it is run on Google's servers which cannot decrypt ESP images. Moreover, server-side classification of images compromises their privacy. Consequently, ESP utilizes client-side image classification, similar to what Apple does on their Photos app on mobile devices, to protect their users' privacy [88]. In fact, both Google and Apple provide on-device classification models, Google via ML Kit [89] and Apple via Core ML [90]. Although ML Kit provides fewer labels for on-device classification compared to Google's cloud-based one, Apple's Core ML has no such limitations. Once labeled, images can be searched.

Users may wish to edit images such as by adding filters or cropping them. Images are edited locally by first decrypting the image, applying the modifications, then re-encrypting the image. ESP can be supported on web browsers via browser extensions. Such an extension implements all the features of a normal ESP client. Depending on local storage constraints, decrypted images may be cached locally. If storage is constrained, then the extension fetches ESP encrypted thumbnails to ensure a smooth user experience.

Existing cloud photo services could change their systems or format requirements in ways that impact ESP users. ESP assumes that services adhere to existing image standards and will not deviate from them, i.e. services will not arbitrarily convert users' photos to different formats. This is unlikely to happen, but if it does, ESP clients would not overwrite their local encrypted ESP images with the copies that the cloud photo service has converted. Since ESP clients keep mappings of photo identifiers, these records can also be used to detect arbitrary modifications by services and retain the copies of the images prior to their modifications, allowing ESP to re-encrypt the images to the new format. Another possibility is that services may release support for encrypted images themselves which would compete with ESP. This is actually a desirable outcome. The

motivation of ESP is to satisfy users' desires for privacy via a client-side solution with no reliance on any third-party services, since cloud photo services do not provide this feature themselves. If services begin to provide features similar to ESP, this would be an overall win for users as the user experience, integration, and feature support can be further improved with support from the services themselves.

4.4.6 Key Management for Multiple Devices

Users do not directly interact with the secret key and seed values used to encrypt images. Instead, they manage self-generated PDK keypairs which are pairs of public/private keys. Unlike E3, which uses PDK keys to directly encrypt user data—emails—as described in Chapter 3, ESP uses PDK keys to encrypt and decrypt a copy of secret key S in the key-encrypting key scheme discussed in Chapter 2. The encrypted S, denoted as $E(S)$, is stored both on users' devices and also remotely in the image hosting service in the form of a QR code image. Then any of the user's devices with ESP can fetch and decrypt $E(S)$ to recover S. The difficulty lies in how ESP should manage keypairs for users, including granting and revoking access to $E(S)$ for a user's multiple devices. As commonly known and shown [13], the concept of pairs of private and public keys is extraordinarily confusing for average users.

PDK, as already shown in E3, addresses these problems by exposing a key management system with a simple usage model from the perspective of users, exposing them only to the concept of device management which they can readily understand. PDK's primary principle is that users do not need to move a private key to and from all their devices, but instead synchronize all their devices' individual public keys. When a user installs ESP on a new device, it generates a new self-generated keypair and uses the PDK verification process to first gain the trust of one of the user's other devices. This already-trusted device then broadcasts the new public key and its other trusted public keys in a signed message encoded in a QR code image uploaded to the photo hosting service. The user's other previously configured ESP clients accept the broadcast since they already trust the announcing device and can verify it via the signature. In this way, adding a new device

builds up a chain of trust in which devices trust each other device and the ones that it trusts as well. Device revocation is handled in a similar way, as a user uses one of his trusted devices to broadcast the removal of another device.

In contrast to E3 which uses the user's email account for its communications channel, PDK uses the image hosting service itself as the channel for its message protocol, with QR codes as the messages. For example in the case of Google Photos, ESP creates a special album named #PDK-QR-CODES which is hidden from the view of the user in the app, and uploads PDK QR code messages to it. We chose QR codes as they can be represented as images which are robust against image compression and resizing. PDK messages are not encrypted by ESP but are signed and verified by every ESP client.

Similarly to E3, users must verify new ESP devices which they perform via a platform-independent verification step that works with any device that has a display screen in which communication messages are conveyed using the image service itself, or via near-field communication (NFC) for devices which support it. The process of adding, and therefore verifying, a new device begins when the user sets up ESP on a new device, and the user then participates in the verification process used in E3 as described in Chapter 3. The existing ESP device, who now has the new device's public key, then uses the PDK protocol as normal to broadcast the new public key.

When a user adds a new ESP device, the existing device that verified the new one decrypts $E(S)$ and re-encrypts S using the new device's public key and all other public keys. The resulting $E'(S)$ is stored as normal which completes the new device setup. Deleting a device is similar: the user deletes it by selecting it from one of his existing devices, then the remotely stored $E(S)$ is re-encrypted using the public keys of all devices except for the deleted one. This simple deletion mode is suitable when the user is still in control of the device being deleted. However, if a user is deleting a device because it was lost, this requires more work since the goal of deleting a device is to ensure that it cannot decrypt newly uploaded encrypted images (a lost device will have access to any locally stored images from before the user deletes the device). We make a distinction between a lost device and a compromised device. ESP does not defend against device compromise

where the attacker has bypassed the hardware or OS-level security because in these situations, the attacker will have access to the user's secret key regardless of the security or encryption scheme. The purpose of device deletion is for a user to revoke a device's access to his uploaded images.

Deleting a lost device consists of replacing $E(S)$ with a new secret key S', and re-encrypting all of the user's images. S' is encrypted using all public keys except for the key of the lost device. The user's other devices will detect this change and update their local caches. The device then begins re-encrypting all of the user's existing images, and continues using S' for any new images so that the lost device will be unable to decrypt them. The existing device which began the deletion request may choose to parallelize the re-encryption effort among other devices by broadcasting how the work should be distributed via PDK's communication protocol. For example, it can partition images by album, time ranges, and more.

In extreme cases, a user may lose all of his ESP devices thus necessitating the recovery of access to his encrypted images. What the user has lost is all of his devices' private keys which can decrypt $E(S)$. Therefore, when a ESP client creates a new $E(S)$ that does not already exist in the image hosting service, it also provides the user with a recovery key that they must write down. The recovery key can be to decrypt $E(S)$, thereby regaining access to the user's images. This means that $E(S)$ is also encrypted using a symmetric crypto key: the recovery key.

4.5 Security Analysis

ESP acts as a significant security barrier to compromises of privacy when users use ESP to encrypt their photos. As the following security analysis shows, it is difficult to break encryption for even one photo. If a user encrypts many photos, the difficulty of breaking the encryption for all those photos becomes almost impossible. This is because adversaries need to expend immense computing resources to break the encryption of even one photo, and more so with many photos since each photo has a unique random seed for its encryption. Even then, the output of attack attempts would need additional resources, either computing or manual human verification, to determine whether the attempts at decrypting photos are correct or not. It therefore becomes

intractable for an attacker to successfuly decrypt many encrypted photos uploaded by a user.

Given significant resources, an attacker might break the encryption of a photo, but it may be more likely that the adversary would attempt to break the encryption via other means, such as stealing a user's trusted device or exploiting vulnerabilities on a user's device. As we discuss in Section 4.2, defenses against attacks like these are orthogonal to the core of ESP design, and would be better handled at the hardware, OS, or even application platform level, i.e. Android and iOS providing mechanisms for secure key management resilient against attackers. Even though ESP considers device security to be out of scope, ESP still represents a significant improvement to cloud photo service security that attackers must overcome as it is much more difficult to obtain users' ESP secret keys compared to the current norm of needing only user account names and passwords to access their photos.

Our security analysis focuses on ESP's two main security guarantees: (1) ESP's device management ensures that photos are encrypted using only authenticated public keys. (2) ESP's encryption protects the confidentiality of photos. We describe ESP's properties to use as building blocks for these claims.

Property 1. *The first configured ESP device is trusted.*

ESP relies on a trust on first use approach since all keypairs are self-generated. This first device's keypair is used as the starting point for adding further devices.

Property 2. *Spurious, unsolicited new device requests constructed by attackers do not result in their malicious devices becoming trusted.*

To compromise a user's trusted device mesh, an attacker could try to force one of the victim's trusted ESP devices to authenticate a malicious one. To carry out this attack, the attacker would need full access to the user's photo hosting service account and initiate the protocol to add a new device. Next, the attacker needs the targeted user to participate in the verification process to authenticate the attacker's malicious device. However, since the user did not initiate the protocol, the user will not recognize the request or know the correct verification phrase, so the malicious device will not be authenticated. Therefore, spurious requests to authenticate malicious devices

will not succeed.

Another approach is for the attacker to wait for the user to legitimately add a new device and then perform a man-in-the-middle (MitM) attack involving the verification phrase, so that the user, who is adding a legitimate new device, is deceived into authenticating the attacker's malicious one. However, this is unlikely, as generally obtaining a new device is a very rare occurrence for users. The attacker may then need to wait an exceedingly long time for the user to add a device; but if the attacker is indeed patient enough, then the attack may succeed.

It is possible to defend against this MitM attack, and therefore require no reliance on the trustworthiness of the photo service's servers, with a modification to the verification process by leveraging a password authenticated key exchange (PAKE) [91]. This approach is implemented and evaluated in Chapter 5 for our third case study, EDP, but we provide a brief sketch of the PAKE-based solution for ESP. Instead of transmitting verification phrases through the photo hosting service, ESP would display the phrase to the user on new device which the user must copy and input onto the trusted device. The verification phrase therefore acts as the password in the PAKE which then allows the two devices to securely authenticate each other. Since the password is transmitted via the user manually, an attacker with access to the user's photo hosting service account cannot intercept the exchange and perform the MitM attack. Therefore, this modification allows ESP and its key management solution, PDK, to be used with even untrusted photo services.

Property 3. *Any device which is trusted by the first ESP device is also a trusted device.*

This follows from Properties 1 and 2. If the first device authenticates a new device's public key, then it is trusted by definition.

Property 4. *Any device trusted by a given trusted device is also trusted by every other device.*

This is the generalized version of Property 3, and holds true due to trust by transitivity. If a device trusts another device, then the first device will also trust every device trusted by the second one.

Property 5. *ESP incrementally builds up trust among all of a user's devices using only a single completed verification process per device.*

In other words, when a user adds a new ESP device, that device does not need to complete the new device verification step with each of the user's existing devices. Instead, it only completes the verification with one existing device. We prove this property via induction. Suppose a user has a single device, then by Property 1 it is trusted. If the user obtains a second device, then the first is trusted, and the second device is verified by the first and becomes trusted. The result is that both devices trust each other and have exchanged their public keys.

Now the user adds a third device and has devices A, B, and C. A and B have already completed the verification process and trust each other. Then, the user runs the verification process with B and C, after which point they trust each other. Since B trusts A, B knows A's public key and provides it to C, so C can also trust A despite having never interacted with it. The question is then how A learns about C's public key. This is achieved by having B publish its set of trusted public keys, signed by itself and containing C's public key, by uploading it to the photo service. A will detect the signed set of trusted public keys, and since A trusts B, it will accept B's signature and therefore accept C's public key. A, B, and C now know every trusted public key and trust each other.

This same logic applies even if the user has N devices and adds a new device Z, resulting in $N+1$ devices. The user adds new device Z by using trusted device $K \in \{N\}$ to verify Z. Afterwards, K and Z trust each other, and Z has received the public keys of all N devices. K publishes an updated set of trusted public keys containing Z's public key, signed by K. The remaining $\{N-K\}$ devices observe Z's new public key in the set, and seeing that it was signed by K, accept it, thereby also accepting Z's public key. In summary, the user has performed only a single verification step for new device Z in order for it to join their ecosystem of trusted ESP devices.

Property 6. *ESP devices that do not participate directly in the verification step add only the new device's authenticated public key to their local key store.*

This follows from Property 4. Trusted devices only accept public key lists from known, trusted devices, so they reject any messages or public keys signed by an untrusted public key.

Property 7. *ESP's device management ensures that a user's photos are encrypted using only the public keys of authenticated devices.*

This follows from Properties 1–6.

Property 8. *ESP's encryption is robust and secure against attacks.*

In this analysis, we examine our ESP-FY design. We first consider brute force attacks both on the secret seed values and images themselves. As described in Section 4.4.2, the seed values (s_R, s_G, s_B) are each 384 bits in length. This means there are $(2^{384})^3$ possible values for (s_R, s_G, s_B), representing a sufficiently large key space. Even if an attacker only wishes to brute force the seed value for a single channel, the complexity is still 2^{384}, and this alone is not enough to reverse the inter-channel shuffle.

It follows that a brute force attack could also be performed on the images. Such an attack must find both the correct permutation of 8×8 blocks and ordering of inter-channel swapped RGB blocks to reconstruct the image. There are consequently $O(B_R! \times B_G! \times B_B!)$ possible permutations of blocks to reconstruct the entire original image. In either case, brute force attacks are impractical for all but the smallest of images. For example even a small (by modern standards) 1280×720 pixel image contains 14400 blocks, for a search space of $14400!^3$.

An extension of the brute force attack is to treat shuffle-based encrypted images as if they were unsolved jigsaw puzzles. A jigsaw puzzle solver attack leverages perceivable outlines within an image's shuffled blocks to try to re-assemble them into recognizable features. The running time of puzzle solving techniques increases exponentially with the number of blocks, and shuffling blocks across color components significantly reduces the output quality of these solvers [92]. Using a small enough block size such as 8×8 pixels blocks also acts as an important defense against puzzle solvers [93]. Even when the solvers run to completion, there is often a low or zero reconstruction rate of the source image's recognizable features.

Attackers may not need the correct position of every block to gain useful information about an image. For example, an image may contain sensitive information in only a small portion of it which a puzzle solver or outline counting attack may reconstruct, or a "close enough" guess can reveal the context of the original image. Since more than 1 billion images were uploaded to Google Photos in 2019 [73], ESP represents a significant barrier to adversaries like the cloud service itself

Figure 4.4: The image and its two ciphertexts that do not pass the NPCR randomness test by a slim margin.

analyzing all of these user images. It is less clear what happens if the adversary is targeting a very few images, e.g. in the context of a specific investigation; this is the subject of ongoing research.

ESP's design is resistant to known plaintext attacks due to its adaptive key scheme: the encryption keys (seed values) used to encrypt every image is different. It is also robust against differential cryptanalysis [94]. We evaluate this claim using a commonly used measurement, the number of pixel change rate (NPCR) [95, 94]. This metric compares two ciphertext images C_1 and C_2 for a source image which has been modified by 1 pixel. The formula for NPCR is:

$$NPCR(C_1, C_2) = \sum_{i,j} \frac{D(i,j)}{W \times H} \tag{4.1}$$

if $C_k(i,j)$ represents the pixel in the ith column and jth row in image C_k. W is the width of the images in pixels and H is the height in pixels, and both are assumed to be equal in images C_1 and C_2. The output value for NPCR is in the range $[0, 1]$.

If $NPCR(C_1, C_2) = 1$, this suggests that images C_1 and C_2 are completely different. This is however an unlikely outcome. Therefore values of $NPCR(C_1, C_2) < 1$ can be evaluated via a randomness test with significance level α. If $NPCR(C_1, C_2) < NPCR*_\alpha$ where $NPCR*_\alpha$ is the critical value of the NPCR test, then $NPCR(C_1, C_2)$ fails the randomness test. This critical value represents an ideally encrypted image indiscernible from a completely random image [94]. The

critical value $NPCR*_\alpha$ is defined as:

$$NPCR*_\alpha = \frac{\text{Pixel}_{\text{MaxValue}} - \phi^{-1}(\alpha)\sqrt{\frac{\text{Pixel}_{\text{MaxValue}}}{W \times H}}}{(\text{Pixel}_{\text{MaxValue}} + 1)}$$

where ϕ^{-1} is the inverse cumulative density function (CDF) of the standard Normal distribution.

We computed the NPCR for 100 images we selected from the Open Images Dataset V5 [86] and encrypting the original and modified versions. Since ESP splits RGB images into three encrypted grayscale images, the inputs C_1 and C_2 were constructed by combining each of their three grayscale ciphertext images into a single RGB ciphertext image. In our tests, 99 of 100 images passed the NPCR randomness test such that $\text{NPCR}(C_1, C_2) > \text{NPCR}*_{\alpha=0.05}$ for each pair of images C_1 and C_2, suggesting that ESP's encryption scheme is substantially resistant to differential cryptanalysis. The last image was very close to passing this test, as shown in Figure 4.4; we believe that further minor improvements to our algorithm will bring it to a passing score.

Property 9. *ESP guarantees the confidentiality of its secret key and secret seed values.*

ESP uses standard and widely used public key encryption for protecting the secret keys which encrypt the secret seed values per image that drive the ESP-FY encryption scheme. The asymmetric public keys from each of a user's devices encrypt the ESP secret key, which is a high entropy, randomly generated symmetric AES key, and this key encrypts the secret seed values for each encrypted image.

Property 10. *ESP's encryption protects the confidentiality of user photos.*

This follows from Properties 8–9.

4.6 Implementation

We implemented ESP on Android by modifying Simple-Gallery, a popular open-source image gallery app used by millions of users [96], and the Android Fresco image loading library [97], which uses the libjpeg-turbo [98] library written in C. Simple-Gallery was originally an offline image gallery app, so we modified it to support Google Photos. We implemented PDK by adapting

a separate Android key management app, OpenKeychain. We implemented the JPEG encryption algorithm in C/C++. We modified Fresco's image pipeline to invoke the encryption and decryption routines when requested.

Since Simple-Gallery had no support for online services, nearly all the added code, about 4.5K lines of code (LOC), was for interfacing with Google Photos. The Java code added to the Fresco library mainly consisted of abstraction layers orthogonal to encryption to accommodate the software design patterns for their image pipeline. The encryption algorithm written in C/C++ only required about 1K LOC, not including the GNU Multiple Precision Arithmetic (GMP) Library [99] for arbitrary precision floating point values.

4.7 Experimental Results

We verified that ESP is compatible with popular photo services, is robust against ML labelers, and measure its overhead. We ran our tests with 2500 JPEG images selected from the Open Images Dataset V5 [86]. The selection process consisted of randomly choosing $N = 2500$ rows from the dataset CSV file, discarding the D_N number with dead links, then selecting $N - D_N$ new images and repeating this process until the number of unique images totaled N. To avoid Google Photos resizing images, ESP resized any greater than 16 MP (4920×3264) using bilinear downsampling, and saved the result as an 85 quality JPEG. All performance tests were executed on a Samsung Galaxy S7 smartphone with a Snapdragon 820 processor and 4 GB RAM on Android 8.0 using our modified Simple-Gallery app retrofitted with the Fresco and libjpeg-turbo libraries. Internet access was via the smartphone's WiFi connected to a Verizon Quantum Gateway G1100 5 GHz WiFi router with a Verizon FiOS 300/300 Mbps residential fiber optic connection.

4.7.1 Compatibility and Interoperability

We ensured that ESP is compatible with popular cloud photo services, namely Google Photos, Flickr, and Imgur. We randomly selected 100 of the 2500 images randomly selected from Open Images Dataset and encrypted them using ESP. For each service, we we uploaded the encrypted

images, waited a period of time to ensure the images were processed by the service, such as applying compression, then downloaded, decrypted and manually imspected each image. In general, ESP images are compatible with any photo hosting service if they are not resized or if the full resolution versions are available. Flickr displays small resized images but also provides links to the originals. Links to full size versions can be used by ESP to correctly decrypt images. Most services have an arbitrary maximum limit on file size or image dimensions but this has little bearing on ESP which can simply resize images to each service's limits. All services apply compression of varying strengths, but images remain compatible with ESP in the sense that they do not suffer from visual artifacts or corruption beyond what is normally caused by JPEG compression. We manually inspected the images on a high resolution display and found that the differences in quality for most images compared to the source images were imperceptible unless we greatly magnified them, and even then it was difficult to say which looked definitively better from a psychovisual perspective; it was more a matter of individual preference. For the remaining experiments, we focused on Google Photos.

For Google Photos, we also confirmed that ESP has acceptable image quality across all 2500 images in our sample set. To provide a quantitative measure of image quality, we measured the peak signal-to-noise ratio (PSNR) to compare the following against the source images: source images processed by Google Photos (*Google Photos*), decrypted ESP images (*ESP*), and ESP images processed by Google Photos before decryption (*ESP Google Photos*). For ESP, we obtained measurements for encrypting images using three levels of JPEG quality, 50, 85, and 100. Figure 4.5 shows the PSNR for each; a higher PSNR suggests that the level of noise in the image is more similar to the original and is therefore of better image quality. *Google Photos*'s average PSNR was 40 dB. *ESP*'s average PSNR were 38 dB for 50 quality, 39 dB for 85 quality, and 40 dB for 100 quality. *ESP Google Photos*'s average values were 36 dB for 50 quality, 37 dB for 85 quality, and 38 dB for 100 quality.

Although ESP necessarily has some effect on image quality due to compressing the grayscale ciphertext images, the PSNRs for both *ESP* and *ESP Google Photos* were not that different from

Figure 4.5: Image quality measured using PSNR relative to source images.

using Google Photos directly, but provide an added security benefit. Industry recommendations indicate that a PSNR between 30 to 50 dB is considered good, with higher being better [100]. On average, both *ESP* and *ESP Google Photos* provide PSNRs within that range. As the results indicate, ESP users may choose higher JPEG quality settings for better image quality as measured by the higher PSNRs for higher levels of JPEG quality.

For *ESP Google Photos*, some images had a PSNR below 30 dB, with the worst case PSNR for any image being 25 dB. This was more common on lower resolution images. We manually inspected the images with PSNRs lower than 30 dB and observed two things. First, the visual quality of these images compared to the source images was not noticeably different from the other images we manually inspected as part of our compability testing. Second, the lower PSNRs occurred for images that could be described as being low quality images in the first place, in the sense that the source images were generally low resolution and blurry. This suggests that the lower PSNRs are unlikely to occur for real photos of interest that are encrypted using ESP and stored using Google Photos.

For some images, *ESP* has a lower PSNR compared to *ESP Google Photos*. In these cases, the noise introduced by ESP's intermediate compression was specifically reduced by Google Photos' processing pipeline, which suggests that they apply a noise filter to uploaded images. ESP clients may therefore use noise filters or similar algorithms to improve the visual appearance of images if an unusually noisy one is detected.

Since one of ESP's threats is the ML classifiers used by cloud services, we ensured that they fail to correctly label ESP images. We ran Google's ML Kit image labeler on our test images and their ESP-encrypted versions. We then compared the labels to verify if any matched. ML Kit labeled the encrypted images with "Pattern" which none of the images contained. Some encrypted images also experienced other false positives and had labels unrelated to the original images.

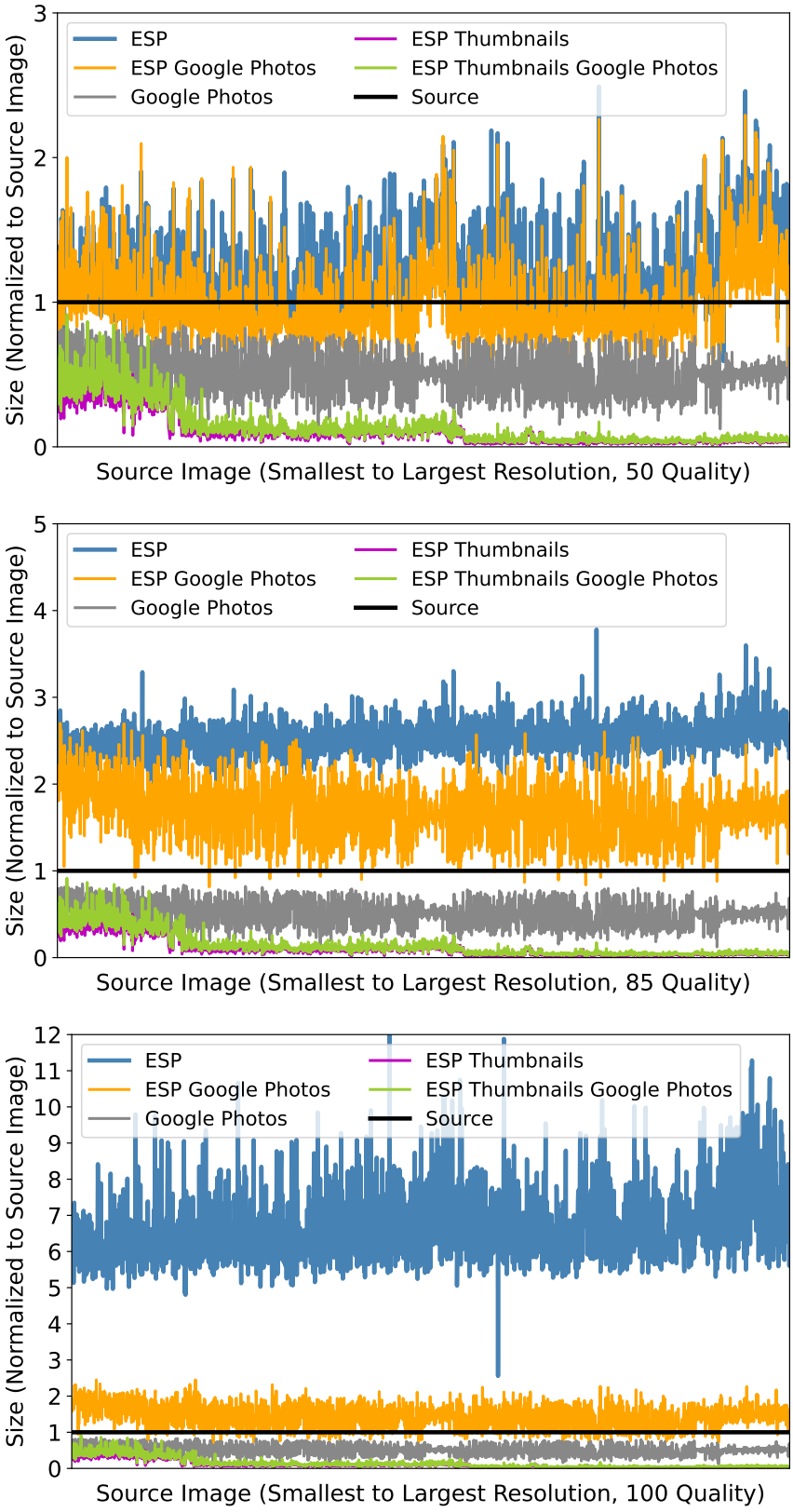

Figure 4.6: Image file size overhead normalized against source images.

4.7.2 Performance Overhead

We compared the performance of using Google Photos directly versus using Google Photos with ESP. First, we compared performance in terms of the image file sizes uploaded and downloaded from Google Photos. For ESP, we used the same encrypted images at the three levels of JPEG quality, 50, 85, and 100, discussed in Section 4.7.1. ESP also creates an encrypted thumbnail for each image, which was done by first scaling source images to the 1/8 factor nearest to a target dimension of 400×400 pixels; resizing JPEGs this way is significantly faster than downsampling them to precisely fit within a 400×400 pixel box. Thumbnails were encrypted at JPEG quality 50. We only created thumbnails for source images larger than 800×800 pixels.

Figure 4.6 shows the measurements for the 2500 images, with file sizes normalized to the respective source image file sizes; smaller is better. When using Google Photos directly, the uploaded image file is that of the source image (*Source*), and the download image file is after Google Photos compresses the image (*Google Photos*). When using Google Photos with ESP, the uploaded image files consist of the encrypted ESP image files (*ESP*) and the encrypted ESP thumbnails (*ESP Thumbnails*), and the downloaded image files consist of the encrypted ESP images after they are compressed by Google Photos (*ESP Google Photos*), and encrypted ESP thumbnails after they are processed by Google Photos (*ESP Thumbnails Google Photos*). Although ESP generates three separate grayscale JPEG images during encryption, we treat them as one and measure the sum total of all the image file sizes.

Using ESP, file size overhead increases with ESP's JPEG quality setting. The average file size for *ESP* was 1.2, 2.5, and 6.5 times the source image file size for 50, 85, and 100 quality, respectively, quantifying the file size overhead for the encrypted image that is uploaded to Google Photos. File size overhead for *ESP Thumbnails* was negligible except for the lowest resolution images since in those cases, thumbnail resolution and file size were no longer insignificant compared to the source images. The file size for *ESP Thumbnails* was on average less than a tenth of the source image size, but in the worst case, it was .6 times the source image file size for the lowest resolution image.

For ESP, other than the JPEG quality setting for encryption, the primary determining factors of the encrypted image file sizes were the permutation of pixel blocks for a given encrypted image and the properties of the source image itself. A shuffled image is unlikely to have sizable regions of consistent color and generally appears close to random noise, thus preventing efficient compression, so encrypted image file sizes are larger. Other factors include properties of the source image, including its original JPEG quality and chroma subsampling format. If the source image itself was saved with a high JPEG quality, i.e. higher than 85, then converting its RGB data to three grayscale images with 85 quality results in greater file size compression. However if the source image is already saved with low JPEG quality, there is little gain from compressing it further, resulting in a larger file size overhead. Similarly, if the source image uses chroma subsampling such as 4:2:0, the CbCr components are a quarter of the size of the Y component but the output encrypted RGB grayscale images are all full size and not downsampled. In contrast, if the source image has full size CbCr components (4:4:4 format), then the output encrypted grayscale images are effectively the same resolution as all of the YCbCr components in the original image, resulting in less inefficiency.

Figure 4.6 also shows the file size overhead for the encrypted image that is downloaded from Google Photos, which is different from the uploaded image because Google Photos compresses images. The average file size for *ESP Google Photos* was .9, 1.7, and 1.4 times the source image file size for 50, 85, and 100 quality, respectively, quantifying the file size overhead for the encrypted image that is downloaded from Google Photos. File size overhead for *ESP Thumbnails Google Photos* was negligible except for the lowest resolution images. In comparison, the average file size for *Google Photos* was half of the source image file size.

The ESP-encrypted images processed by Google Photos are sometimes larger than the ESP images before they are processed by Google Photos. One explanation for this phenomenon is an apparent oversight by Google Photos' image processing and compression pipeline at the time of writing. The original unprocessed encrypted grayscale images output by ESP are true grayscale JPEGs with only one color component, the luminance (Y) channel. However, Google Photos

seemingly processes all images as if they are color JPEGs with the YCbCr colorspace. In other words, Google Photos converts true grayscale JPEGs (Y) to color JPEGs (YCbCr), and needlessly populates the Cb and Cr components with the luminance data. Although Google Photos also forces its output images to use 4:2:0 chroma subsampling meaning that the CbCr components are downsampled, they still represent extraneous overhead. An optimization for ESP's encryption would be to output each encrypted grayscale JPEG as a color JPEG while only keeping the useful data in the Y channel, and populating the Cb and Cr channels with zeros.

Although source images are typically not saved with JPEG quality 100, the measurements suggest that this quality setting may be useful for ESP because Google Photos appears to more aggressively compress the large JPEG images with 100 quality compared to the lower JPEG quality settings. For *ESP Google Photos* the average file size overhead for 100 quality was surprisingly less than that for 85 quality. The main downside to using the 100 quality setting would be that it could take much longer to upload the photos to Google, since the average file size to upload is much larger for 100 quality than for 85 quality. However, using 100 quality would not need to consume much more local storage space if only the original unencrypted images are retained locally. Note that the file size overhead can greatly vary depending on the properties of the source image. For example, one *ESP* photo in the graph showing JPEG quality 100 size overhead has a distinctly lower file size of about 2.4 times the source image file size, compared to the average of 6.5 times, because it is a high resolution photo of the night sky with nearly no stars visible, making it an almost solid black JPEG photo.

Next, we compared performance in terms of the time to upload to and download from Google Photos. For these measurements, we used the 100 images randomly selected from the 2500 image sample set which we originally used for manually testing compability. When uploading an image, ESP first concurrently encrypts the source image and a thumbnail, which results in three separate grayscale JPEG images for the image and three more for the thumbnail, which are then concurrently uploaded to Google Photos by invoking a Google Photos API to register the uploaded images to Google Photos as Google media items. We measure the entirety of ESP's encryption

and uploading time for all six images together (*ESP (Upload)*). When downloading an image, ESP separately downloads and decrypts the encrypted images and thumbnails, so we can measure their respective download and decrypt times (*ESP (Download)* and *ESP Thumbnails (Download)*) separately. For comparison, we also measure the time to upload the unmodified source image to Google Photos (*Google Photos (Upload)*) and download the respective image from Google Photos after it has been compressed and processed (*Google Photos (Download)*).

Figure 4.7 shows the upload and download times for ESP for 50, 85, and 100 quality, respectively. Although ESP's upload and download times are slightly higher with higher quality, the difference is small, suggesting that, at least for a fast residential Internet connection, the choice of JPEG quality setting for ESP should be based on factors other than upload and download times.

ESP upload and download times are larger than directly using Google Photos, which is not surprising given the added encryption and decryption costs and the fact that the image files being transferred between client and server are also larger. Nevertheless, the difference in both upload and download times between directly using Google Photos and using ESP is at most a few seconds in all cases, though the difference is larger for uploading than downloading. Although the encrypt and upload times are larger than the download and decrypt times, encrypting and uploading occurs in the background and is not in the critical path of the user, who is free to continue using the app and perform other actions. This usage model is no different from the official Google Photos app, which also does background uploads. ESP's higher upload and download times, especially given that uploading can be done in the background, is arguably worth the additional security benefit it provides.

Figure 4.7 also shows the upload and download times (*ESP (Upload w/o Encrypt)* and *ESP (Download w/o Decrypt)*) without including the time to encrypt and decrypt the grayscale JPEG images, respectively. Comparing *ESP (Upload)* and versus (*ESP (Upload w/o Encrypt)*, we can see that most of the time is spent on uploading rather than encrypting, though encryption costs as a percentage of the total time increases at larger image resolutions. On the other hand, comparing *ESP (Download)* and versus *ESP (Download w/o Decrypt)*, we can see that most of the time is

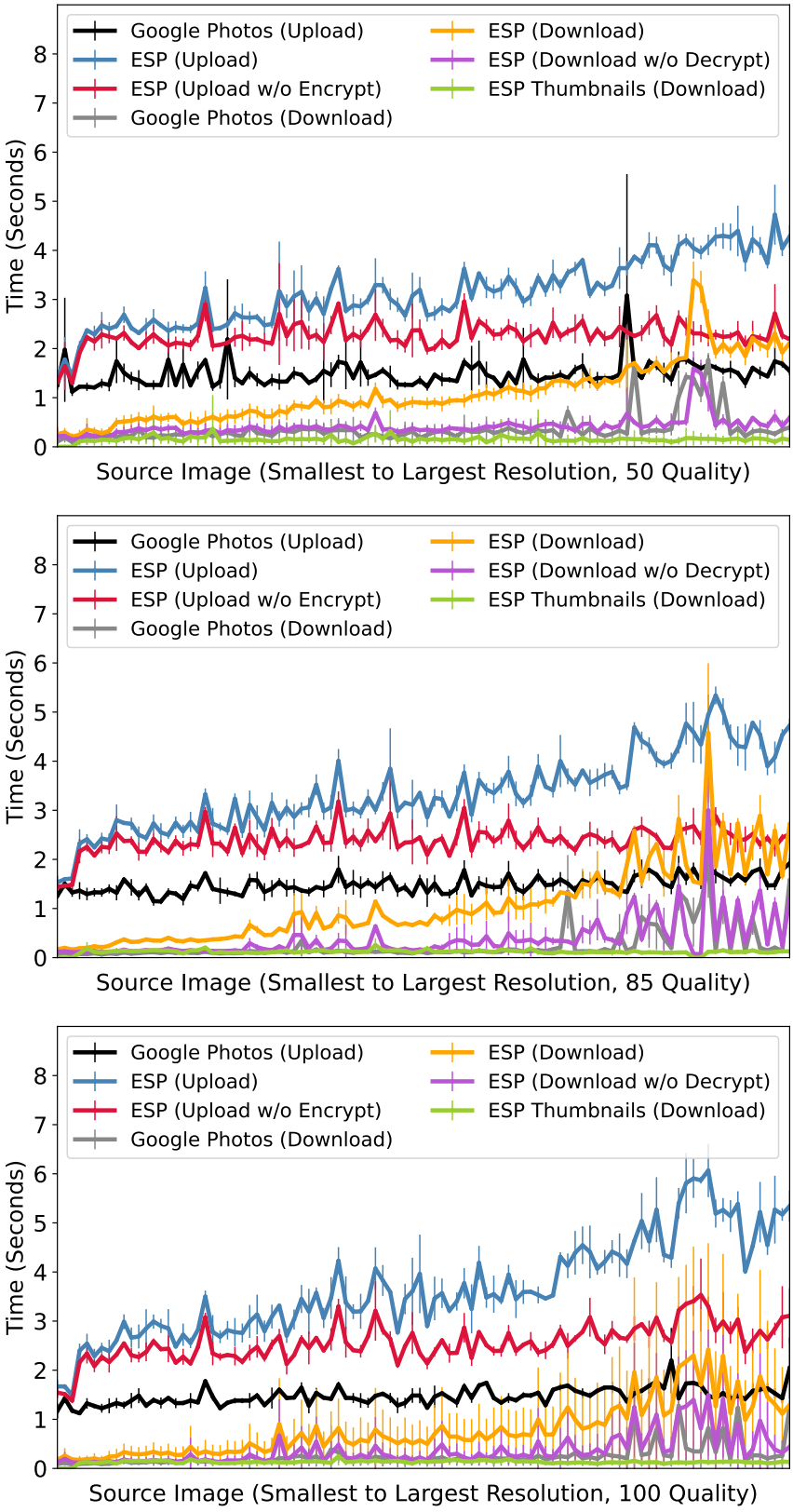

Figure 4.7: Image upload and download times using Google Photos.

Mean	Stdev.	Min.	Q1	Median	Q3	Max
79	18	43	67	83	92	100

Table 4.1: System Usability Scale summarized scores.

spent on decrypting rather than downloading. While downloading the encrypted images is not much different from downloading the processed source image from Google Photos, it is so fast that decrypting the full size images adds significant overhead.

Downloading and decrypting ESP thumbnails (*ESP Thumbnails (Download)*) is faster than downloading unencrypted images from Google Photos on average. This highlights the importance of leveraging encrypted thumbnails; they exhibit far lower overheads, generally less than 250 ms, to download and decrypt. ESP stores images locally in plaintext after decrypting them so downloading ESP images and thumbnails are a one-time cost per device, but the usage of encrypted thumbnails is still a critical point for providing a smooth user experience when browsing newly synchronized images. A cleverly implemented app can aggressively fetch and decrypt thumbnails, and lazily fetch full size images as the user selects them. Then adjacent images can also be fetched and decrypted in the background. The user will then only notice loading times if swiping quickly through full size images that have not yet been cached locally on the device. The overhead can also be eliminated from the user's perspective if images are preloaded and decrypted locally ahead of time in the background while the user is looking at an image or the app is idle.

4.8 Usability

ESP's daily operation is transparent to users: they have the same experience as with a regular image or photo app. The primary difference in ESP is new device configuration, which includes setting up the first ESP device and adding any others. Either setup case requires the only significant new interaction from the user compared to a normal app. We consequently performed a small pilot user study of ESP with a focus on the configuration steps which are necessary for key management.

We administered a user study approved by our institutions' IRB (protocol number AAAS8276)

with 18 study participants who used the PDK system for ESP. Due to the government and IRB enforced prohibition of human interaction in response to the on-going COVID-19 pandemic, we were unable to recruit more users. Participants were allotted 60 minutes but none used the entire time. Twelve were 20 to 29 years old, four were 30 to 39, one was 40 to 49, and the last was 50 to 59. We asked them to setup the app on two of their personal Android devices when possible, otherwise we provided them with either a Samsung Galaxy S7 and a Huawei Honor 5X, or an LG K9 and Google Pixel 2 XL. Having subjects use their own devices when possible was to help avoid biases caused by users becoming frustrated with an unfamiliar device; this is an issue unrelated to our study aim. We supplied Google accounts which we helped users set up on their devices so that they could select these accounts to use for Google Photos. Users set up the app on one device and completed any required configuration steps. Next, they were asked to repeat this identical process on their second device and then perform the verification step. In our study, users used the verification phrase method. All users finished in 5 to 10 minutes.

Finally, users completed the System Usability Scale (SUS) [61], an industry-standard survey used to evaluate the overall usability of a system. The SUS scores are summarized in Table 4.1; a higher score correlates with better usability. For example, a score of 85 to 100 suggests *Excellent* usability, 71 to 85 means *Good*, and 51 to 70 means *OK* [101]. Although ESP's median SUS score is 83, in fact 9, or half, of the users gave SUS scores of 85 or higher and therefore rated the system as having *Excellent* usability, while the remainder felt it had between *OK* and *Good* usability. Only one user gave a score lower than 51; however this user expressed disinterest in the concept of photo security and encryption which may have biased their survey choices. Other user comments such as "That was easy" during the sessions suggests that the primary usability overhead of ESP, new device configuration, is simple and intuitive.

4.9 Related Work

Researchers have tried many different approaches to encrypting images, especially JPEGs. Much of this research in the fields of computer vision, and signal and image processing is inspired

by [102] who first introduced the idea of using chaotic maps such as the logistic map to drive image encryption. Many have tried using different kinds of chaotic maps such as Arnold's cat map [95, 103, 104, 105, 106], or combining several maps to generate improved distributions [107]. However, all of these suffer from the limitation of either assuming a raw bitmap without accounting for inefficient JPEG compression, or they modify and permute DC and AC coefficients in ways that break JPEG compression algorithms.

Early work describes the concept of format-preserving encryption (FPE), where an encryption outputs a ciphertext which retains the formatting and length of the original plaintext [108]. This was extended to full files, such as applying FPE to the problem of encrypting images [109, 110]. A common approach is to scramble JPEGs within the constraints of its format by modifying DCT coefficients with different approaches such as directly obscuring [111, 112, 113] or scrambling them [114]. Some designs have the express intent of specifically preserving JPEG image file sizes [115, 116, 117]. There are countless methods for secure JPEG scrambling schemes [112, 116, 113, 117, 107], but they make no claims about compatibility with existing cloud photo services. Our independent implementations and tests of these classes of encryption show that they are in fact incompatible with services such as Google Photos.

Works designed for cloud storage often break the JPEG format and therefore rely on a third-party service exclusively tailored for their ciphertext format [78, 79, 80, 81, 82]. Others introduce unreasonably large performance overheads [103, 118, 119, 106] or sacrifice significant image quality [118]. Some build on format-preserving encryption by not only encrypting the original image but also outputting recognizable encrypted thumbnails [118, 119], but suffer from performance issues, while others encrypt only specific regions of interest (ROI) within images to obfuscate identities or sensitive material [78, 79, 80, 81, 82]. These works tend to encrypt the ROIs of an image, extract them from the remaining unencrypted parts, and store them separately in either generic cloud storage offerings or their own servers. These approaches therefore are not compatible with existing cloud photo services such as Google Photos.

The security of ROI encryption is not well understood [119]. In lay terms, the privacy guaran-

tees are unclear because there are no well-defined models for judging whether an encrypted ROI protects users. An ROI approach could define ROIs as human faces and obscure them, but the remaining visible portions of the image may yield sensitive information such as location, time, relationships, et cetera. Furthermore, some solutions ask users to themselves select the ROIs in an image, which not only is tedious but also unreliable as users do not understand the privacy and security implications of ROIs. Although not strictly an ROI-based approach, a recently developed system called Fawkes [120] allows users to "cloak" their uploaded photos to shield them against facial recognition software, which ostensibly is also effective against ML labelers in general. Fawkes' strategy resembles ROI approaches in some ways, as it focuses on obscuring human subjects' faces rather then encrypting entire images. Fawkes therefore suffers from potential privacy issues from other threats beyond facial recognition software since environmental and contextual information is left uncloaked. One of Fawkes' important claims is that the cloak consists of changes at the pixel level which are imperceptible to the human eye, but independent evaluations observe significant unwanted visual modifications to photos [121]. In contrast to Fawkes, ESP encrypts entire photos, thereby shielding users against any kind of adversary, with negligible effects on visual quality.

ESP's encryption algorithm is inspired in part by a previous encrypt-then-compress strategy [93], but that approach suffers from two significant problems that make it unworkable for use with cloud photo services. First, it is not compatible with services such as Google Photos because it modifies pixel values in a way that results in corruption after compression. Second, it results in a massive increase in file size and a 3× increase in image dimensions. The increase in image dimensions of their ciphertext images reduces users' effective maximum upload dimensions for Google Photos from 16 MP to 5.3 MP. ESP introduces a new encryption algorithm which is compatible with Google Photos' compression and does not have file and image size problems, avoiding prematurely bumping into the 16 MP Google Photos limit. ESP also introduces other key features to support cloud photo services, including supporting encrypted thumbnails and sharing and end-user key management.

Most approaches do little, if anything, for key management. The few works which do [78, 119, 110] only briefly suggest that the secret (private key or password) used to decrypt images on other devices—the user's own or another person's—for viewing should be distributed via a secondary out-of-band channel which is never specified or evaluated.

4.10 Summary

Easy Secure Photos (ESP) makes it possible for users to encrypt their images and use them with existing cloud photo services such as Google Photos, thereby providing privacy against cloud providers and other adversaries. ESP achieves this with purely client-side modifications by introducing a format-preserving encryption method for JPEG images and a unique key management solution which leverages the cloud photo service itself rather than any third parties. Moreover, the encryption method is compatible with compression algorithms used by real services such as Google Photos. We have implemented ESP by integrating it in an existing Android photos app and evaluated its security, performance, and usability with existing cloud photo services such as Google Photos. Our results show that ESP (1) is compatible with popular cloud photo services, including Google Photos and Flickr, (2) is resistant to various attacks including differential cryptanalysis, (3) maintains good image quality for encrypted images even after being processed through Google Photos' image processing and compression pipeline, (4) incurs only modest overhead on upload and download times when used with Google Photos, and (5) is easy to use as encryption and decryption is transparent to users, setting up a device to use ESP is simple, and everyday usage of ESP is no different from a regular photos app.

Chapter 5: Easy Device-based Passwords (EDP)

5.1 Introduction

Popular media [122, 123, 124] and government institutions [125, 126] often recommend that people use password managers for greater security and convenience in managing passwords to access online services. Managers provide greater security by generating strong, unique, complex, and easily changed passwords for online accounts, avoiding the problems caused by common, duplicate, and short passwords used across multiple user accounts. Managers provide greater convenience by only requiring users to remember one master password for the manager, instead of needing to remember many different passwords for many different online accounts. Commercial password managers are increasingly popular; three well-known options, 1Password, Dashlane, and LastPass, serve over 40 million private users and hundreds of thousands of business entities [127].

While password managers protect user accounts with strong computer-generated passwords, these passwords are protected by weak human-created master passwords. These master passwords are more easily guessed, and since they protect many other passwords, they are a prime target for attackers to harvest user credentials for many online accounts. Two-factor authentication (2FA) is increasingly used to provide additional protection beyond weak master passwords, but this only mitigates the problem of attackers compromising a password manager account. More problematic is that managers encrypt the file used to store a user's passwords with a key derived from the weak master password. Despite services using key-strengthening schemes such as PBKDF2, the resulting derived keys are usually not difficult to brute force [3, 4, 5], and even if they are hard to guess, they often protect information that is valuable enough to be worth the effort and resources needed for brute forcing [128]. Any attack on the password manager servers that exposes the encrypted password file makes it easy for attackers to then decrypt that file and harvest user credentials on

a large scale. Unfortunately, it is hard to evaluate the security operations of these services since most commercial offerings' infrastructure and password databases are black boxes and security via obscurity is prevalent [129]. The result is that users must blindly trust password managers and the commercial services providing them, hoping that server security operations are good enough to sufficiently safeguard users' encrypted password files at rest. In fact, such blind trust is unwarranted as commercial services have detected potential leaks and hacks involving their users' encrypted password files [130, 131, 132]. To exacerbate the problem, there are also potential privacy concerns as information sent to password manager servers may be used to track and analyze what websites users are accessing.

To address this problem, we have created Easy Device-based Passwords (EDP), an easy-to-use password management system that protects password files using public key cryptography. EDP retains the concept of master passwords to authenticate users, but further requires that an authenticated user must be using a trusted device to gain access to the user's passwords. The device used to initially setup the user's password manager account is designated as trusted. Any additional devices can only be designated as trusted with the approval of an existing trusted device. An attacker who obtains the user's master password but is not using a trusted device cannot access the user's encrypted passwords.

EDP enforces the use of trusted devices using public key cryptography, in a new way that avoids usability problems associated with it by applying the PDK design. Each trusted device generates its own private/public keypair, then EDP encrypts a user's password file using the public keys of all trusted devices such that it can be decrypted using the private key of any trusted device. Because public keys are used for encryption, this greatly increases the security of encrypted passwords at rest as the entropy of public keys is many orders greater than that of human-usable master passwords. EDP thereby eliminates the risk of brute force attacks on password files at rest. Because password files are encrypted by a trusted device with the device's own public key before they are sent to password manager servers, users do not need to make any assumptions about the quality of the security operations of password manager services. Instead, users can treat these services

as simple untrusted data stores that provide network infrastructure and redundancy guarantees. Furthermore, EDP preserves users' privacy since all encryption of password files occurs client-side, so that no information about users' browsing habits is revealed to service providers.

A key aspect of EDP is its support for multiple trusted devices. EDP has a new secure bilateral authentication scheme that differs from and is an improvement to the security of the verification process described in Chapters 3 and 4, allowing devices to securely exchange public keys, and communicate updates for purposes such as adding and revoking devices. Adding a device is done via a device pairing mechanism that simply requires the user to view a random word phrase on a trusted device and enter it on the device being added. Since keypairs are self-generated, there is no need to expose any key exchange or PKI concepts to users.

While EDP supports the use of multiple trusted devices, it does not require users to use multiple devices to access their passwords; a single trusted device is sufficient. EDP takes an approach that is fundamentally different from and more secure than just 2FA. 2FA often requires obtaining a code from a trusted smartphone device separate from the device on which the user may be accessing the password file, thereby requiring multiple devices to access passwords. In contrast, EDP provides strong public key encryption of password files while only requiring users to use a single trusted device to access password files.

Because EDP does not require any security guarantees for the servers, many different types can be used. For example, a conventional web server or a shared, cloud-resident file system could be used. We present a decentralized EDP backend architecture that enables its use with existing peer-to-peer (P2P) and distributed database (DDB) infrastructure. We introduce a P2P discovery mechanism to enable a user's devices to discover each other without requiring any form of user login account. We also describe how to use a DDB for storage of encrypted password files.

We have implemented a prototype of EDP by modifying the open source Bitwarden password manager [133], used by millions of users. Our modifications disable the use of Bitwarden's servers and consist only of client-side changes. In lieu of Bitwarden's servers, our prototype stores users' encrypted password files in a decentralized and distributed database, and uses P2P connections to

securely perform automatic and transparent PDK key management among users' devices, including exchanging public keys, adding and verifying new devices, and removing devices. Our security analysis shows that EDP guarantees the confidentiality of password files and detects any compromise of the integrity of password files. Our user study and performance results demonstrate that EDP is easy-to-use and performs comparably to stock Bitwarden while providing much stronger password security.

EDP is the first password manager that does not rely on server-side security. It does require secure clients—but in fact, all password managers do. EDP ends reliance on secure servers by using high-entropy encryption keys and a strong peer-to-peer authentication system for adding new devices, facilitated by a novel easy-to-use public key cryptography system.

5.2 Threat Model

We assume that any services or servers that EDP uses are untrusted and the administrators and infrastructure supporting them may be malicious. We assume that EDP clients and users' devices on which they run are secure and trustworthy. We do not protect users' devices because that is the responsibility of the hardware or operating system (OS). A compromise of a device implies that the attacker has access to any private keys on the device, rendering most user mode defenses fruitless. Even if such private keys are encrypted using an unlock password, hardware or OS support is necessary to deter offline attacks since generally unlock passwords memorized by humans are easy to guess. If a lost or stolen device is a concern, then remote device wiping features are a reliable defense. We assume that any encryption algorithms we use do not contain backdoors to break the encryption. Attackers may attempt offline attacks on the encrypted data used in EDP, but we assume that the security of any standard public key encryption systems that we leverage are vetted and robust against such attacks. This is an orthogonal concern to EDP.

5.3 Usage Model

EDP is designed to be easy to use and provide a familiar usage model similar to existing password managers. A first-time user simply needs to install an EDP app on one of the user's devices and set up an account by providing a username and master password. The EDP app then provides the user with a recovery key which should be printed or written down somewhere safe in case the user ever forgets the master password. Once the user confirms that this has been done, EDP's setup is complete. The EDP password manager app may be used in exactly the same manner as any other password manager, such as remembering user-inputted passwords, filling in password fields, synchronizing passwords across the user's devices, and adding and removing synchronized devices.

Users should only install and configure EDP on devices they trust, such as their own personal smartphones, tablets, and computers. Installing a password manager on only trusted devices is a generally accepted usage model for any password manager, not just EDP. This is because using a password manager on an untrusted device such as a public computer represents a severe security risk despite any defensive measures that try to protect users' secrets. EDP does not support usage on untrusted devices such as those at an Internet cafe or public locations.

Users may install and configure EDP on multiple devices and then use any of them to manage their passwords. There is no limit on the number of devices that users may configure for use with EDP. Each new device only requires a one-time brief, platform-independent verification step with one other previously configured device. The verification step is presented to users as a device pairing process where a user verifies the new device with any of the user's already configured EDP devices. This is done by simply having an existing EDP device display a random phrase, which the user then enters on the new device, thereby designating the new device as a trusted EDP device for the user. All of a user's trusted devices are known to each trusted device. A user may use any trusted device to remove any other device from the set of trusted devices. This does not necessarily prevent the device from using old stored passwords since it may have stored them locally, but does

prevent the removed device from learning any new or updated passwords after it has been removed.

EDP leverages the fact that many users own multiple devices and is designed as a password management solution for multiple devices. However, EDP does not require a user to have access to more than a single trusted device to perform any of its operations. EDP can continue to operate even if a user loses all but one of his devices with no special recovery steps; he may still access and manage his passwords on that one remaining device as long as he also remembers the master password. If the user loses all his devices and remembers his username, master password, and recovery key, he may recover his passwords by configuring a new EDP device and providing those three pieces of information. In the worst case, a user may lose all his devices, forget his username and master password, and lose his recovery key, in which case, like nearly all secure password management solutions, the user will be unable to recover his passwords. However, most online services offer their own password recovery or reset mechanisms. In this situation, the user can perform the password recovery or reset steps for each online service for which the user wants to recover or reset the respective password. In other words, the password manager serves as a password cache for online services to make it easier for users to access them, but if the cache becomes unavailable, it is generally inconvenient but possible to recover access from the online services themselves.

5.4 Architecture

To understand how EDP works, we first briefly describe a typical password manager service architecture, as shown in Figure 5.1. A user selects a username and master password on the client. The client derives a key by applying a PBKDF to the master password, uses the derived key to encrypt the user's password file, and uploads the file to the server such that the username can be used to locate the file. Any device can be used to access the password file by providing the respective username and master password. The user trusts the password manager server to safeguard the weakly encrypted password file.

In contrast, the EDP architecture, depicted in Figure 5.2, requires no such trust in the password

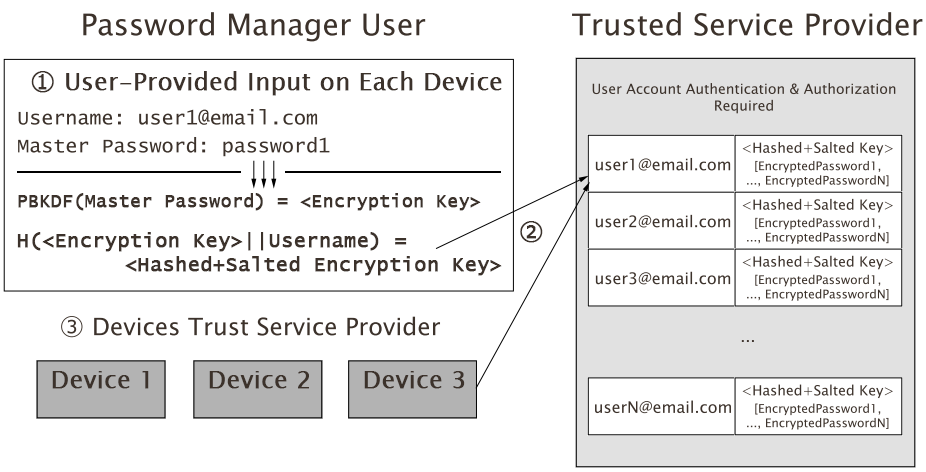

Figure 5.1: The architecture for a typical password manager service.

manager server, but instead only requires a user's devices to trust each other. A user selects a username and master password on the client. The client generates a keypair, uses the public key to encrypt the user's password file, and uploads the file to the server such that the username and master password can be used to locate and retrieve the file. The username and master password are never used for encryption, only to generate (1) an index key to locate the password file for that user, and (2) a communication channel key to create a private communication channel among the user's devices. Only a user's trusted devices can be used to access the password file by providing the respective username and master password. Any updates or modifications which do not originate from the user's configured EDP devices are immediately detectable. The user does not trust the password manager server and relies on public key cryptography to safeguard the strongly encrypted password file, guaranteeing its confidentiality and integrity. EDP cannot ensure the availability of users' encrypted passwords stored on an untrusted server—an adversary could simply delete users' passwords on the server—but the availability guarantee is no different from any existing password manager service.

While using public keys to encrypt password files greatly improves their security compared to existing password managers, public keys have historically been fraught with usability issues [13] as discussed in Chapters 2, 3, and 4. EDP avoids these usability problems by using PDK, or in

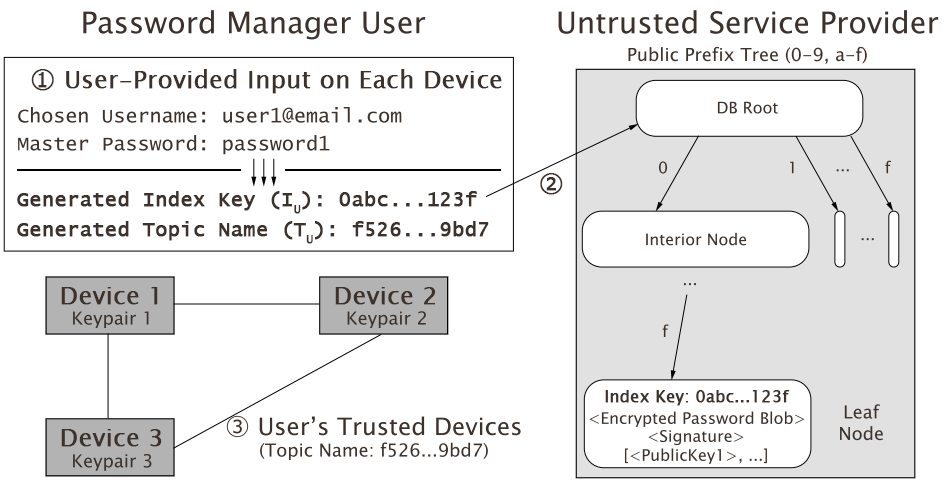

Figure 5.2: The EDP architecture for users and providers.

other words, self-generated keypairs that are entirely internal to the system so that users never need to know about public/private keypairs. Just as in E3 and ESP, each of the user's devices generates its own keypair. But in EDP, the public keys from all of the user's devices are used to encrypt the user's password file. As discussed in Chapter 3, PDK eliminates the troublesome issue of needing to educate users on how to securely transfer private keys because the private keys never leave the device on which they were generated. The remaining problem is how the user's devices can securely exchange public keys among themselves with minimal intervention by the user, so that any single device can encrypt and decrypt his passwords. This must be done such that only trusted, or authenticated, public keys that are verified to belong to the user are exchanged so that an adversary cannot trick EDP into encrypting a user's password file with a malicious public key. This issue of identifying trusted keys is normally handled via PKI by having trusted authoritative servers sign certificates, but this is a tedious and confusing process so EDP specifically avoids any reliance on PKI.

As in both E3 and ESP, EDP's PDK instantiation also uses self-generated keypairs that must be authenticated in some manner to be considered trusted. Since there is no external root of trust, EDP ensures that all of the user's devices trust each other, and this chain of trust is constructed in the same manner as E3 and ESP: the user unknowingly participates in an authentication step when

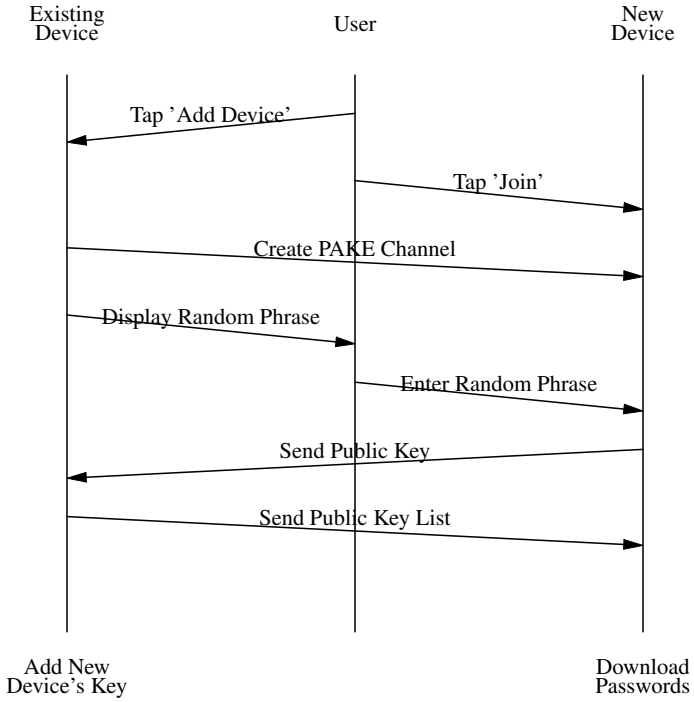

Figure 5.3: Adding a new device in EDP.

adding a new device, which is added to the set of trusted devices. However, EDP introduces a significant improvement to the authentication process used in E3 and ESP which allows the use of an untrusted communication channel, in contrast to E3 and ESP which used secure, trusted connections. The user can also provide the correct username and master password on any authenticated device to remove other devices. Every time the user adds or removes a trusted device, the resulting membership set is uploaded to the password manager server as a signed message, so that the user's other devices can learn about the changes.

Encrypted password files are also signed by the device that did the last password change. This signature can be validated by any device that downloads the file.

5.4.1 Trusted Device Authentication

EDP's device authentication protocol, summarized in Figure 5.3, allows a user to add a new device to their set of trusted EDP devices by communicating with one of their existing EDP devices.

The purpose of this design is to ensure that a user with an existing EDP device can authenticate and add a new device's public key to its list of trusted public keys with cryptographic certainty that it has accepted the correct public key, and not one belonging to an attacker. The device authentication protocol's usability hinges on the ability of EDP to construct a secure communication channel between a user's existing EDP device and the new one that he is adding with minimal input from the user. This involves two steps, having the devices agree on a communication channel and then securely encrypting that communication channel. A variety of communication channels are possible, but, for simplicity, we will describe these two parts assuming the devices communicate over the Internet.

To have the devices agree on a communication channel, EDP makes use of some well-known server as a relay server to connect devices across different network topologies, including across NATs and firewalls, for example. The basic requirement is that both devices must independently agree on some shared identifier for the communication channel; if they can agree on an identifier, they can use this as a rendezvous mechanism. We refer to this identifier, a communication channel key, as a topic name T_U for user U. For example, if a user's devices both present topic T_U to some sort of relay server, the server knows that these devices are all interested in the same topic, and therefore are looking for each other and should be connected together.

An EDP client computes T_U when the user initiates the process of adding a new device on one of his existing EDP devices. To make the topic name unique and easy for a user's devices to agree upon but harder for an outside attacker to identify, we define the topic name T_U as

$$T_U = H(KCH_{KDF(username_U||password_U)}(username_U)||\text{“}DeviceChannel\text{”})$$

where KDF is a key derivation function, KCH is a keyed cryptographic hash function, and H is a hash function that provides a standardized output format, following a standard approach for generating a cryptographic hash. KDF generates a secret key K_I using the user's username and master password as inputs into a standard key derivation function such as bcrypt, scrypt, or PBKDF2.

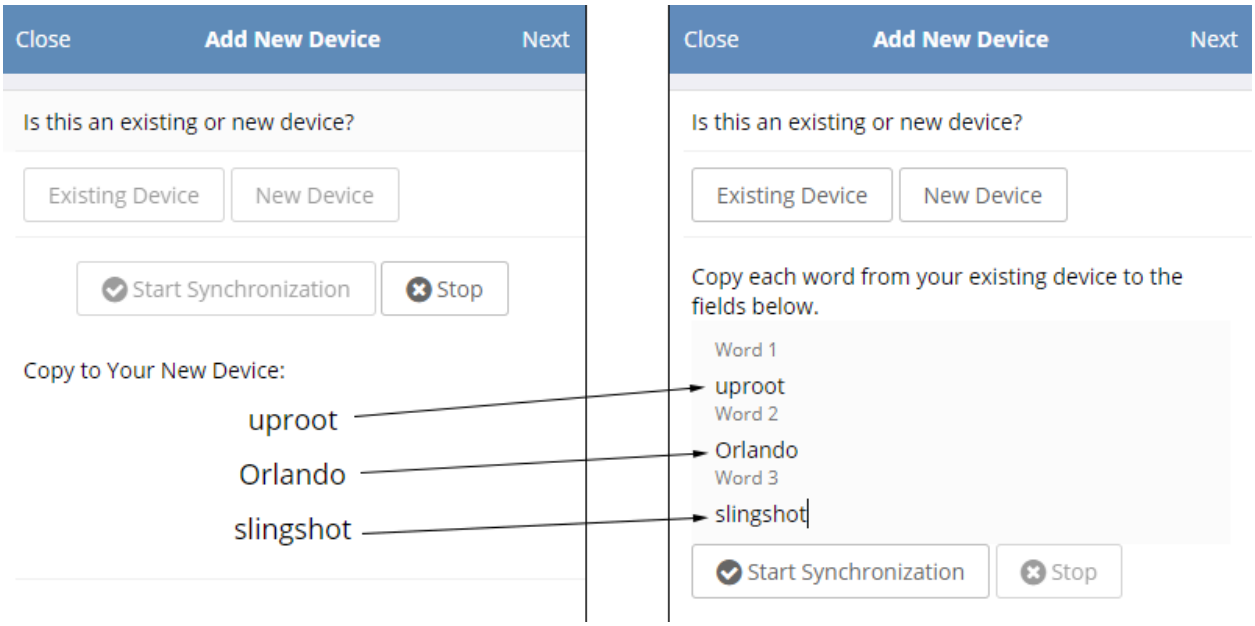

Figure 5.4: Users copy a random phrase from their existing trusted device to their new one.

KCH_{K_I} is a standard keyed cryptographic hash function that takes the secret key K_I and some data and has been vetted for making it difficult for an attacker to guess the secret key and data inputs [134]. The string "Device Channel" is a label which acts to expand the keying material and identify the use for the key. EDP uses the popular keyed cryptographic hash function HMAC-SHA256 with the username as the data; many other options are possible. H is a SHA-256 hash function to standardize the output format and length regardless of the chosen hash algorithm for the keyed cryptographic hash function KCH_{K_I}. By using T_U, EDP devices can discover each other independently through a relay server without the need to establish any kind of login in advance, thereby establishing a communication channel for the device authentication protocol between a new device and an existing EDP device.

To secure the established communication channel between the new device and an existing EDP device, the devices perform a password authenticated key exchange (PAKE) [91]. A PAKE protocol uses a shared secret, often a simple password, to authenticate a cryptographic exchange. The result is a shared session key and proof that each party knows that secret. Furthermore, the messages contain no verifiable plaintext [135] that would allow an attacker to validate a guess at

the secret. In other words, guessing attacks on intercepted messages are not possible. Recall as discussed in Section 5.3, when the user initiates the process of adding a new device on one of his existing EDP devices, the existing EDP device displays $RandomPhrase_N$, an N-word random phrase, to the user, which the user then enters on the new device. This is depicted in Figure 5.4. EDP uses $RandomPhrase_N$ as the password to generate the secret key K. In other words, we leverage the user as a secondary, out-of-band channel. The user, who has physical access to the two separate devices, manually but unknowingly provides enough information for the devices to use PAKE to authenticate each other by means of having copied $RandomPhrase_N$ from the existing EDP device to the new device. Both devices therefore know $RandomPhrase_N$ without ever having communicated it over the communication channel, but can prove to each other their knowledge of it without disclosing it. For $RandomPhrase_N$, EDP uses an N-word phrase that is easy for humans to identify, copy, or select from a list. It is selected at random when the device authentication protocol is initiated. For usability purposes, EDP uses a N of 3 to 5 random words chosen from a curated list such as the PGP Word List [52]. It is also possible to substitute numbers instead of words, for example 3 to 5 randomly generated sets of 4-digit numbers. EDP clients select a new $RandomPhrase_N$ every time a user initiates the device authentication protocol.

The details of the PAKE are depicted in Figure 5.5. After the trusted and new devices connect to topic T_U and discover each other, they a PAKE based on the Diffie-Hellman key exchange (DH) [136] to secure their communication channel by encrypting it with a secret key K and to exchange public keys. The usage of a PAKE is important given that EDP operates using untrusted servers; DH key exchanges are normally unauthenticated, but EDP uses PAKE so that both devices independently generate a secret key K. Both devices therefore know $RandomPhrase_N$ without ever having communicated it over the communication channel, but can prove to each other their knowledge of it via a zero-knowledge password proof, so they know any communications between them are authentic. As shown in Figure 5.6, knowledge of both T_U and $RandomPhrase_N$ by a given device is needed to become a trusted EDP device within the user's EDP ecosystem so an attacker cannot masquerade as a new device that a user is adding. By confirming knowledge of

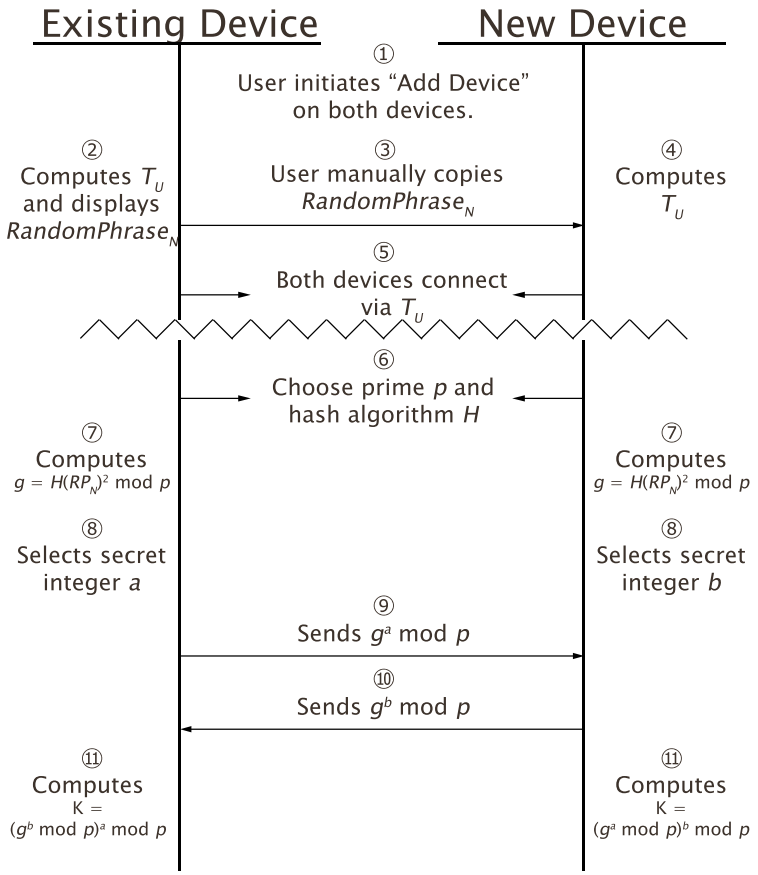

Figure 5.5: EDP's PAKE-based device authentication protocol to compute secret key K to encrypt communications.

these two pieces of information, EDP can ensure that the existing device and the new device belong to the user.

After having authenticated each other, the devices exchange their public keys over the secure communication channel encrypted by K, including any public keys that they already trust. This is notably done without the user needing to know about public key cryptography. The final result is that the existing and new devices have exchanged public keys and trust each other. And since the existing device sent its list of trusted public keys, the new device also trusts those devices' public keys via transitive trust.

The user may have other trusted EDP devices which were previously configured, but did not take part in the device authentication protocol with the newest device. These other devices may

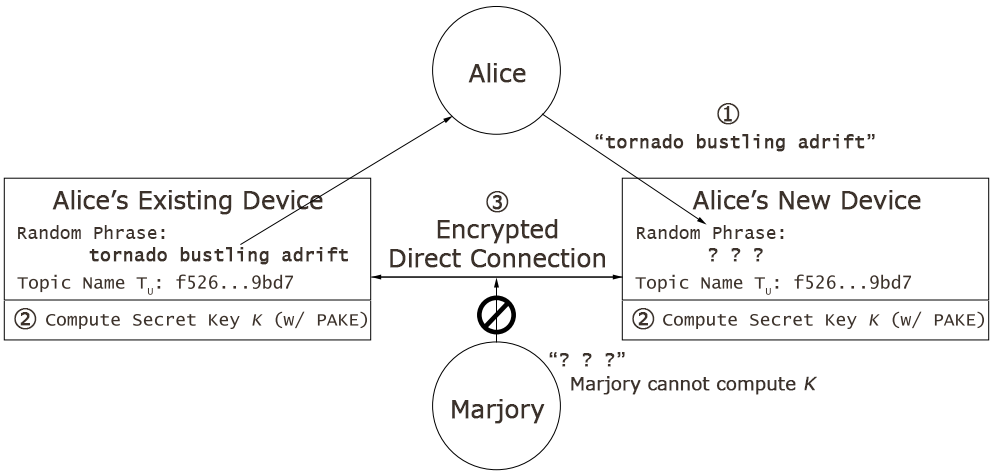

Figure 5.6: EDP's device authentication protocol requires T_U and $RandomPhrase_N$ to compute secret key K.

also be offline for an indefinite period, and possibly may never be online at the same time as the two devices the user used in the device authentication protocol. EDP therefore requires a method for devices which have not participated in the device authentication protocol to learn about newly added or revoked devices. This is done by having the existing EDP device, which did participate in the protocol, publish a signed and timestamped list of its trusted public keys together with the user's encrypted password file; the latter is described in Section 5.4.2. The user's other devices can then observe updates to the list; since these are signed by a device they trust, they accept the update that contains the new device's public key. Each EDP device checks for the list and accepts updates before performing any operations that involve updating the set of trusted devices or encrypting the user's password file. If the signature for the trusted public key list does not belong to one of the user's trusted devices, the list cannot be trusted and is ignored. The trusted device which detected the untrusted list replaces it with its own list of trusted public keys, signed by itself.

Users may lose one, some, or all of their devices. If a user loses any number of devices but not all of them, the user can use one of the remaining EDP devices to remove all of the lost devices from the set of trusted devices. The device used for revoking the lost devices publishes a signed and timestamped list of its updated trusted public keys along with a re-encrypted password file; the

update can then be obtained by the user's other EDP devices. By removing the lost devices, the user is ensured that the encrypted password file is re-encrypted using the public keys of only the remaining trusted devices so that the lost ones cannot access and decrypt the password file. As an extra precaution, the user may also choose to select a new master password and change all of his passwords. If the user loses all of his devices, he can configure EDP on a completely new device and provides his username, master password, and the EDP recovery key which he recorded on his first setup. Providing this information to EDP allows it to locate the user's encrypted password file and decrypt it with the recovery key.

5.4.2 Password File Operations

A password file contains a user's saved passwords mapped to their services or domains. The exact format of the file is implementation specific; it is common for password managers to use their own formats for their encrypted passwords and to support importing the password file formats of competing services. However, the format should satisfy two requirements: (1) each password is timestamped at the time the user updates it and saves it locally, and (2) deleted passwords are marked with a tombstone marker at the same time. The timestamps and tombstone markers enable EDP devices to determine whether their local copy contains stale password data in the presence of updates on multiple devices and network failures. For example, consider a user with devices A and B, where A has lost network connectivity. The user may update a password entry on A which is saved locally but cannot be updated at the server. The user may later change his mind and update the same entry on B, making A's entry stale. The timestamps allow the devices to identify which password is the most recent one to use. Devices sign all password files, and thus passwords, timestamps, and tombstone markers can be verified. Although we refer to the concept of a singular password file, a user's encrypted passwords may in fact be encrypted and stored separately. The decision of whether to store all passwords in a single ciphertext or multiple is implementation dependent. If they are separately stored, then each individual password entry with any metadata including its timestamp and tombstone marker are signed.

To protect the password file, a given EDP device generates a random 256-bit AES key and encrypts it with the public key of each authorized device. This password file is cached by the device and stored on an EDP server, where it can be accessed by any of the user's EDP devices. If a user uses an EDP device to update the password file, such as adding or removing passwords, the EDP device checks the server to see if it has a more recently updated password file, in which case it downloads the password file from the server. The device runs a simple timestamp-based resolution algorithm to merge the local cached copy and the one from the server; passwords with a newer timestamp, including the tombstone marker, take precedence. Because updates to passwords are initiated by human users who operate at the granularity of tens of seconds or even minutes, standard automatic time synchronization used by devices is more than sufficient and any remaining clock skew is unlikely to be an issue. After completing the merge, the client then sends the updated password file to the server so that the user's other EDP devices can detect the updated password file and update their local caches. Polling is done to detect updates and reduce the likelihood of updates being in the user's critical path. Any password file update is signed by the respective EDP device. If the signature for the password file does not belong to one of the user's trusted devices, the password file has been tampered with and cannot be trusted. The trusted device which detected the untrusted password file replaces it with its own cached password file, signed by itself.

To locate the password file, and the list of trusted public keys, in a privacy-preserving manner which decouples usernames from users' password files, EDP uses an index key I_U for user U. The key is the hexadecimal-encoded, SHA-256 hash of the result of a keyed cryptographic hash function KCH_{K_I}, K_I being the secret key derived from the username and master password:

$$I_U = H(KCH_{KDF(username_U||password_U)}(username_U)||\text{“}PasswordStore\text{”})$$

which is similar to but distinct from the hash used for the topic name T_U. Because I_U and T_U are distinct, even if an attacker knows one of them, it remains difficult to correlate it with the other. I_U is used for locating user U's password file and is not used for any cryptographic operations

such as encrypting passwords. Due to the use of a keyed cryptographic hash function, commonly used for message authentication codes (MAC), an adversary cannot determine a given user's index key without also knowing the user's master password. Even with knowledge of the master password, an adversary cannot decrypt a user's password file but also must have compromised one of the user's devices to obtain a private key which can decrypt the password file. If an EDP client stores passwords separately instead of in a single file, it addresses an individual password I_{U+D} for domain D with:

$$I_{U+D} = (KCH_{KDF(username_U||password_U)}(username_U||D)||\text{“}PasswordStore\text{”}).$$

A user may wish or need to change his master password. Changing the master password also changes the user's index key I_U. This is invisible to the user but initiates several steps behind the scenes: (1) the user's EDP device downloads the existing password file if it is more recently updated than its local cached copy, (2) computes the new index key I'_U based on the new master password, and (3) stores the password file and its list of trusted public keys at I'_U on the server. If a master password is changed due to a device compromise, the EDP client should also generate new passwords for all of a user's credentials, so that there is little risk even if the old password file is compromised by attackers.

5.4.3 Back End Services

EDP only has basic availability requirements for its communication channel for device authentication and its database for password storage. There are no requirements for confidentiality and although it is good if they can maintain integrity, integrity failures are detectable by users' devices. Communication channels and password stores may be implemented in many different ways. For example, a traditional centralized server, such as a web server, can orchestrate a communication channel among a user's devices by acting as a relay server, and also receive and transmit encrypted password files to store in a typical web server database. Alternatively, a cloud-resident shared file

system could also be used. Ordinary network links, e.g., via Bluetooth or on-LAN connections, are an obvious way to implement a communications channel if the environment permits. However, direct Internet links between two devices probably do not work, given the prevalence of dynamic IP addresses, firewalls and NATs, etc. This is thus not suitable for users who may wish to trust a work and a home desktop. In contrast, we opted for a more decentralized and distributed solution using P2P networking for device-to-device communication channels, and a DDB for storing passwords.

EDP runs its device authentication protocol over device-to-device connections constructed using P2P networking. This is done in a manner reminiscent of BitTorrent's P2P file-sharing protocol's concept of magnet URIs, also known as magnet links [137]. The idea of magnet links is that a piece of data can be identified and addressed by its hash value by leveraging a distributed hash table (DHT). In BitTorrent, a client can use a magnet link—a hash unique to the files the users wish to share—rather than a Torrent file to discover other peers sharing the files. Magnet links can be used together with BitTorrent trackers, servers that help peers discover each other, or by purely relying on the DHT for peer discovery. EDP repurposes this idea not for sharing files, but for creating P2P connections for communication purposes among EDP devices. EDP devices independently generate the topic name T_U for a given user U, a hash that can be used as a magnet link, to find other interested peers, namely a user's devices configured with EDP. EDP can then leverage existing BitTorrent-based P2P systems for setting up communication channels between devices, even in the presence of dynamic IPs, firewalls, and NATs. Although a P2P system may use relay servers that are not securely controlled by a password manager, this is not an issue for EDP because it can be used with untrusted relay servers.

EDP uses a DDB to store users' encrypted password files and encrypted trusted keys. The DDB stores password files and keys in leaf nodes of prefix trees keyed by the index key I_U for a given user U. Instead of storing a single password file at I_U, our EDP implementation stores passwords separately per domain such that the password file for each domain D is addressed by I_{U+D} which are direct children of the I_U node. The root node is globally addressable via a well-defined default

key so that any EDP client can locate it. Interior nodes encapsulate each possible prefix value in the hex-encoded I_U. Since each interior node represents a prefix $p \in [0-9, A-F]$, each node, starting with the root node, has 16 child nodes, one for each hex value. Thus, a EDP client with a given I_U traverses the prefix tree according to the hex values in the index key to locate the associated leaf node with the encrypted password file for user U. The DDB can also provide replication of leaf nodes for reliability and performance. Although a DDB can potentially store data on nodes that may not even be securely controlled by a password manager, this is not an issue for EDP because of how the password file is protected. Many DDBs [138, 139, 140, 141, 142, 143] can be used to support EDP's prefix tree.

5.5 Security Analysis

We distance ourselves from a common attitude towards password manager security focused on defending against local [144] or web-based attacks that leverage vulnerabilities in client implementations [129]. We consider these concerns orthogonal to the focus of EDP, which is to provide a system with strong encryption for storing passwords and easy key management that can be adopted by any password manager, independent of whether the client implementation has exploitable bugs.

We make a two-part claim concerning EDP's security: (1) EDP's method for device management, mainly through its device authentication protocol, is a secure way to authenticate users' devices and public keys that is robust against attacks even when using untrusted servers and unencrypted communications channels. (2) EDP protects passwords at rest with greater security guarantees than typical password managers even with a more difficult threat model allowing for storing passwords on untrusted servers. We describe EDP's properties which we use as building blocks for these claims.

Property 1. *The first device a user configures with EDP is a trusted device.*

This is similar to a trust on first contact security model since EDP uses self-generated keypairs. This first keypair is what becomes associated with a user's password manager service account, and is used as the starting point for adding further EDP devices via the device authentication protocol.

Property 2. *It is infeasible for an attacker to guess the correct length-N random phrase in the device authentication protocol without being detected.*

Since EDP authenticates new devices via an interactive process initiated by the user, the only time an attacker can exploit it, even with knowledge of the master password and length-N random phrase, is the exact moment when a user is adding a new EDP device, which is likely a rare occurrence. If the attacker catches a user performing the protocol, the attacker could try to guess the length-N random phrase. Using a 512 word list [52] and N varying between 3 and 5, the phrase has an entropy of 27–45 bits, well within the capabilities of even a modest attacker. However, the EDP client detects failed attempts to complete the PAKE, generating a new phrase for each new attempt. Furthermore, the guessing rate is limited by the user's retry rate, several seconds at least. Guessing the phrase is thus sufficiently unlikely that it does not pose a realistic threat; the attack is not feasible.

We can quantify this. Assume that a user is trying to add a new device, while the attacker wishes to add a fraudulent device. Suppose that we want the probability of a successful attack to be $\frac{1}{1{,}000{,}000} \approx \frac{1}{2^{20}}$, and that we are using a length-3 phrase, i.e., with an entropy of 27 bits. It will take on average 2^{27} tries (since we are sampling with replacement, given that a new phrase is generated for each attempt) to guess the proper phrase. The attacker can achieve the successful attack probability with only 2^7 tries. This may seem too easy until you realize that the user will be notified of each failed attempt and will have to restart the authentication process, something that will likely take a few seconds at least, even without any realization that something might be wrong when failures occur.

There are two further defenses that can be employed. First, the trusted device can impose additional delays after the first few failed attempts: "More than 10 failed device-add requests; please try again in 10 minutes." Second, a longer phrase can be used; with 5 words, an attacker would need 2^{25} tries to guess the phrase with a probability of $\frac{1}{2^{20}}$.

Property 3. *EDP is robust against spurious, unsolicited new device requests constructed by attackers.*

The general attack would be for an adversary to try to force a user's EDP device to participate in the device authentication protocol with a malicious device, without the user knowing it. This is not possible unless the attacker manages to initiate the process on the user's own device, and somehow either observes the length-N random phrase on the device to input into the fraudulent device, or tricks the user into providing it. While deceiving the user is not impossible, it represents a targeted attack against a specific user in which either the user's device is already compromised or the user is susceptible to spear-phishing attacks and social engineering. In the latter case, such a user would be even more vulnerable using other password managers as an attacker could more easily trick the user to give away his master password.

Property 4. *EDP's device authentication protocol guarantees the integrity and confidentiality of the messages used to add a new device, even on unencrypted communications channels mediated by even untrusted relay servers.*

The basic design of the device authentication protocol is to construct a device-to-device connection between the user's existing trusted and new device, perform a PAKE to agree on a secret key, then using the secret key to create an encrypted communication tunnel. The construction of the device-to-device connection can be mediated by an untrusted relay server, because all the messages exchanged by the two devices in cleartext only contain data which is safe to reveal to the public. For example, the only data which the PAKE protocol transmits over the channel in cleartext are public primes and large exponentials. The only sensitive data transferred between the two devices is their public keys, but these are only conveyed after the encrypted tunnel has been created from the independently computed secret key via the PAKE, which guarantees the confidentiality and integrity of any data transferred over the tunnel. Therefore, the device authentication protocol can operate on untrusted channels even if they are mediated by untrusted relay servers. Availability is a separate concern; a malicious relay server could simply refuse to connect the user's two devices, but this does not compromise the user's trust model.

Property 5. *EDP's device authentication protocol ensures with cryptographic certainty that the public key provided by a newly added device is the correct one.*

This follows from Properties 2 to 4 by using the user as a secondary out-of-band channel to convey the length-N random phrase from the user's trusted device to the new one, and applying a PAKE to ensure that the trusted device is exchanging messages with the correct new device belonging to the user. The public key can only have been provided by the correct new device, and is therefore the correct one.

Property 6. *Any device which is trusted by the first EDP device is also a trusted device.*

This follows from Property 5. If the first EDP device authenticates a new device's public key with cryptographic certainty that it is the correct one belonging to the user, then the first device can trust the new device, creating a single-link trust chain.

Property 7. *Any device trusted by a given trusted device is also trusted by every other device.*

This is the generalized version of Property 6, and holds true due to trust by transitivity. If a device trusts another device, then the first device will also trust every device trusted by the second one.

Property 8. *EDP incrementally builds up trust among all of a user's devices with only a single completed device authentication protocol per new device.*

We prove this property via induction. Suppose a user has a single device, then by Property 1 the device is trusted. Now consider a user with two devices: the first device is trusted, and the second device completes the device authentication protocol with the first device and becomes trusted. Both devices now trust each other having directly exchanged public keys with each other. Now consider a user with three devices, A, B, and C. A and B complete the device authentication protocol and trust each other. B and C then complete the protocol, so they trust each other. Since B trusts A, B provides A's public key to C, so C can also trust A despite having never interacted with it. The question is then how A learns about C's public key. This is achieved by having B publish a list of trusted public keys, signed by itself, containing C's public key. Since A trusts B, it will accept B's signature on the list of trusted public keys and therefore accept C's public key. A, B, and C now know every trusted public key and trust each other.

This reasoning holds for a user with N devices adding a new device, resulting in $N+1$ devices.

The user adds a new device Z by completing the device authentication protocol with trusted device $K \in \{N\}$. After the protocol, K and Z trust each other, and Z has received the public keys of all N devices. K publishes an updated public key list containing Z's public key, signed by K. The remaining $\{N - K\}$ devices observe the updated public key list, and seeing that it was signed by K, accept it, thereby also accepting Z's public key.

Property 9. *EDP devices that do not participate directly in the device authentication protocol to add a new device add only the new device's authenticated public key to their local key store.*

This follows from Property 8. Since trusted devices only accept public key list updates from other trusted devices, they will never accept an update signed by an untrusted public key, because only trusted devices can add a public key to the list.

Property 10. *The effects of an attack on the availability of the infrastructure used by EDP are no worse than those for a typical password manager service.*

An attacker may perform a DoS attack on the infrastructure used by EDP which will render some functions, such as adding and revoking devices and updating passwords, unusable. However, the primary feature of auto-filling password forms remains unaffected, as EDP caches users' encrypted password files locally, like other password managers. Existing devices cannot retrieve changed passwords, nor can devices being added obtain the encrypted password file, if the necessary infrastructure is being DoSed. The same is true for other password managers.

A potential issue arises in the particular case where an attacker compromises a user's device while performing a DoS attack on the EDP infrastructure, thus preventing the user from revoking the compromised device. If the compromised device cannot be revoked, then the user's devices will continue encrypting password files using the compromised public key. This situation is actually a non-issue. The user will update his master password and encrypted passwords to new ones, and also revoke the compromised device locally, but this revocation will not be synchronized to the other devices since the infrastructure is unavailable. However, since the user has changed all the passwords, the attacker's copy of the user's encrypted password file becomes outdated. As soon the attacker ceases the DoS attack, the user's device then synchronizes the revocation request and

new encrypted password file that excludes the compromised device, so the attacker cannot decrypt it. This assumes that the user changes all the passwords quickly enough, before the attacker can use them. This is no worse than the situation for a typical password manager.

Another attack on availability that a user may perform is based on forcing a user's devices to delete its passwords. An attacker may gain access to a user's device, perform a DoS attack on the infrastructure to gain enough time to guess the user's master password, decrypt the user's passwords using the device's private key, then cease the DoS attack and upload an empty password file. The user's other devices will then synchronize the empty password file, effectively locking the user out. However, this attack fares no better in a typical password manager as they are vulnerable to the same attack. With a typical password manager, an attacker who has access to a user's synchronized passwords can delete them, causing the deletes to cascade to the user's devices. One simple defense is for devices to remember deleted passwords for a limited period of time, say 15 days, allowing a user to recall them if necessary.

Property 11. *EDP's method for device management, mainly through its device authentication protocol, is a secure way to authenticate users' devices and public keys that is robust against attacks even when using untrusted servers and unencrypted communications channels.*

This follows from Properties 1–10.

Property 12. *Encrypting passwords with public keys is much more secure than doing so with symmetric keys derived from human-generated passwords using password-based key derivation functions such as PBKDF2.*

EDP uses public key cryptography in the standard way: the asymmetric public keys encrypt high entropy, randomly generated symmetric AES keys, and these AES keys encrypt users' passwords. This by definition represents an encryption system with much higher entropy and security guarantees compared to keys derived from weak, human-generated passwords run through even 100,000 rounds or more of PBKDF2 or bcrypt, as is commonly done by the majority of password manager services.

Property 13. *EDP guarantees the confidentiality and integrity of users' encrypted password files.*

The claim of confidentiality follows from Property 12 due to the use of public/private keypairs with much higher entropy than PBKDF keys. EDP guarantees integrity in the sense that EDP clients can detect any unrecognized modifications to encrypted password files since they are not only encrypted but also signed using trusted public keys. If such modifications are detected, any of a user's trusted devices can restore the file with the last-known good version.

Property 14. *EDP can store users' encrypted password files in even untrusted servers.*

Properties 12 and 13 show that EDP ensures the confidentiality of encrypted password files. The question is then whether the security is sufficient enough that the encrypted password files can be safely stored in an untrusted server that is potentially publicly accessible. The general consensus of the security community is that time-tested and battle-hardened public key cryptosystems such as RSA and ECDSA are trustworthy. Furthermore, public key cryptosystems are used ubiquitously in public contexts; the vast majority of the Internet now uses TLS/SSL and SSH for banking and other sensitive data uses. If public key cryptosystems were too insecure to be used to encrypt even publicly available information, then the entire Internet, not just EDP, would be at high risk.

If quantum computers are developed and deployed, today's public key algorithms will become insecure. Although we have not implemented it, EDP could easily be enhanced to support a public key re-key operation, where a device's old (and now insecure) key is deleted and a new key for a post-quantum algorithm is used instead. Since each user's EDP environment is specific to that user, there is no need to coordinate the conversion with other users. Similarly, there are no problems with expiring root CA [145].

Property 15. *A compromise of a user's EDP master password does not risk the confidentiality of any encrypted passwords.*

EDP passwords are encrypted using the public keys generated on each of a user's devices, and not using keys derived from the user's master password. Therefore, a compromise of a master password does not risk the confidentiality of any of the user's passwords. It however does give the adversary access to the user's account on the password manager service provider. The consequences of this will vary depending on the service, but should never yield access to the user's

passwords.

Property 16. *It is infeasible to derive a user's master password from index key I_U.*

Due to use of a keyed cryptographic hash function in the computation of I_U, it is intractable for an attacker to derive the secret key used as input to the function. Even if an adversary were to determine the master password, e.g., by a guessing attack, this provides no aid in decrypting a user's encrypted password files.

Property 17. *EDP protects passwords at rest with greater security guarantees than typical password managers even with a more difficult threat model allowing for storing passwords on untrusted servers.*

This follows from Properties 12–16.

5.6 Implementation

We implemented a EDP prototype by modifying the open source Bitwarden password manager [133]. We modified the Bitwarden web browser extension, which is used with all popular web browsers, including Google Chrome, Mozilla Firefox, Opera, Apple's Safari, and Microsoft Edge. Our modifications amounted to roughly 2K lines of code (LOC) including HTML templates for UI elements, but excluding the libraries we added; the entire Bitwarden browser extension is roughly 140K LOC.

For the device-to-device communication channel, we used Hyperswarm [146], a peer connection-based distributed networking stack that implements the BitTorrent P2P protocol. Hyperswarm relies on bootstrap servers, similar to BitTorrent trackers, to help peers discover each other; we used bootstrap servers provided by the Hyperswarm developers.

In lieu of using Bitwarden servers for storing password files, our prototype uses GunDB [138, 139, 140], a widely-used decentralized graph database with millions of users; for example, the Internet Archive uses GunDB. GunDB relies on super peers operated by volunteers to assist with replication and availability guarantees; we used super peers provided by the GunDB developers. We use GunDB to create a decentralized and distributed prefix tree for users' encrypted password

files and lists of trusted public keys.

For public key cryptography, we used RSA because it is already used in Bitwarden with key lengths of 2048 bits as the default setting. We encrypted passwords using 256-bit AES keys using the CBC mode of operation with HMAC-SHA256 authentication. We used PBKDF2 with SHA-256 for our KDF with a minimum of 10,000 iterations, although the number of iterations for our applications of PBKDFs is not vitally important. We implemented encrypted password files as JSON objects encrypted using a random AES key; it in turn is encrypted with all of the trusted EDP public keys known to the device. For the EDP device authentication protocol, we implemented a PAKE aptly named Simple Password Exponential Key Exchange (SPEKE) [147], which is widely used and has been vetted and improved over the years by security experts [148, 149].

5.7 Experimental Results

We present some experimental results measuring the performance and usability of EDP based on our prototype Bitwarden implementation. We quantify the performance of our prototype versus vanilla Bitwarden for various password manager operations, including inserting passwords into forms, updating a password file, and adding a new device. We also conducted a modest Institutional Review Board (IRB) approved user study to measure the usability of EDP with respect to adding a new device, since this is the primary difference between EDP and any other typical password manager in terms of usability.

5.7.1 Performance

We measured performance using the latest available version of the Google Chrome web browser, version 83, running on Windows 10 machines, one with an Intel Core i7-8700K 3.7 GHz CPU with 32 GB RAM and another with an Intel i7-6700HQ 3.5 GHz CPU with 16 GB RAM, both connected to a 1 Gbps residential fiber connection. Measurements involving one client were done using the first machine only.

We first quantified the performance of the primary password management functionality, in-

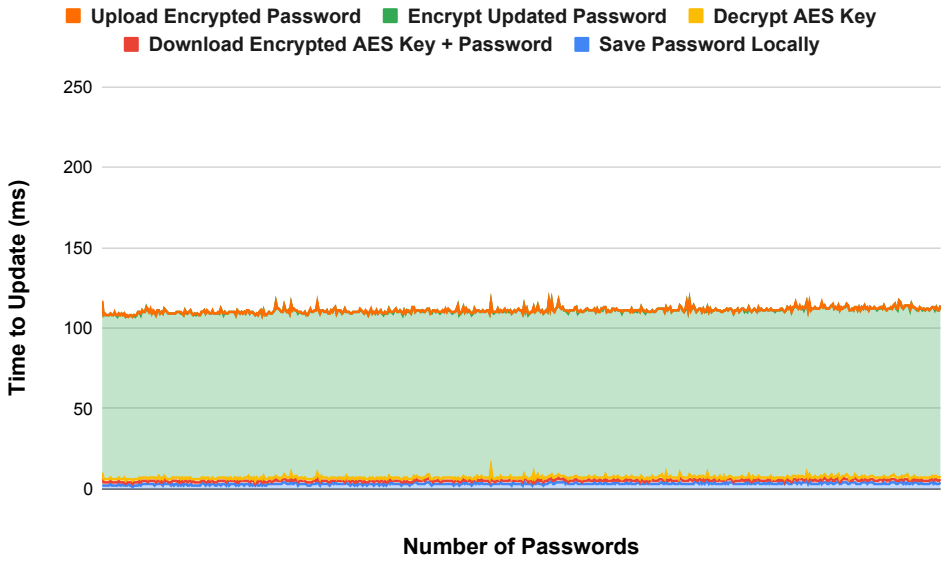

Figure 5.7: Breakdown of the time spent when adding a new password to EDP.

serting passwords into forms. We measured the time to auto-fill password forms with both EDP and unmodified Bitwarden. The average time was the same for both systems, 2 ms, reflecting the fact that EDP makes no changes to any of the Bitwarden code related to this functionality and the time to fill password forms is fast enough to not be noticeable to users. We focus the rest of our discussion on aspects of EDP which did require modifications to the Bitwarden client.

We next quantified the time for a user to add a new password and update the encrypted password file in the password store. We measured the time for both EDP and unmodified Bitwarden, which are shown in Figures 5.7 and 5.8, respectively, for encrypted password files with 1 to 1000 passwords, the latter representing an upper bound on the number of passwords users have in practice. EDP has a constant update time per password while the Bitwarden client's time to add a password scales linearly in the number of passwords. For EDP, we measure the latency of adding a new password and updating the password file to include: (1) locally saving the new password to add, (2) downloading the encrypted AES key and existing password entry, (3) using the device's private key to decrypt the AES encryption key and existing password entry, and check the timestamp and tombstone markers, (4) encrypting the new password, and (5) uploading the encrypted

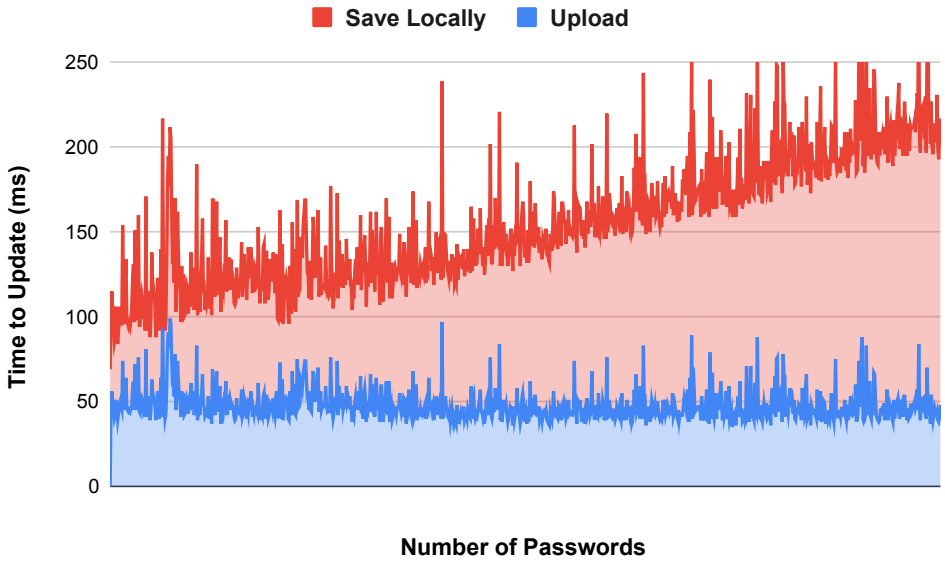

Figure 5.8: Breakdown of the time spent when adding a new password to Bitwarden.

password to the password store. Figure 5.7 shows that most of the time is due to encrypting the password, about 104 ms on average. In contrast, the total time on average is approximately 112 ms. The time to download data is fast due to the benefits of using a DDB, which offers low latency network requests when data is available on nearby peers. While we conservatively included the time to download the AES key and existing password entry being updated, both are typically cached locally, which would further reduce the latency. Because our EDP prototype stores passwords individually as opposed to all together in one password file, updates can be done in constant time. We confirmed that an alternative implementation storing all passwords in one file such that the entire file needs to be downloaded and decrypted for any password update would cause the costs to scale linearly with the number of passwords.

Figure 5.8 provides a less granular breakdown of the time for Bitwarden to add a new password and update the password file. The upload time is relatively constant with the number of passwords because Bitwarden also only sends the additional password on the update, not the entire set of passwords. However, Bitwarden locally maintains a single complex JSON file of all encrypted passwords, and operations on that file are more expensive with more passwords, accounting for the

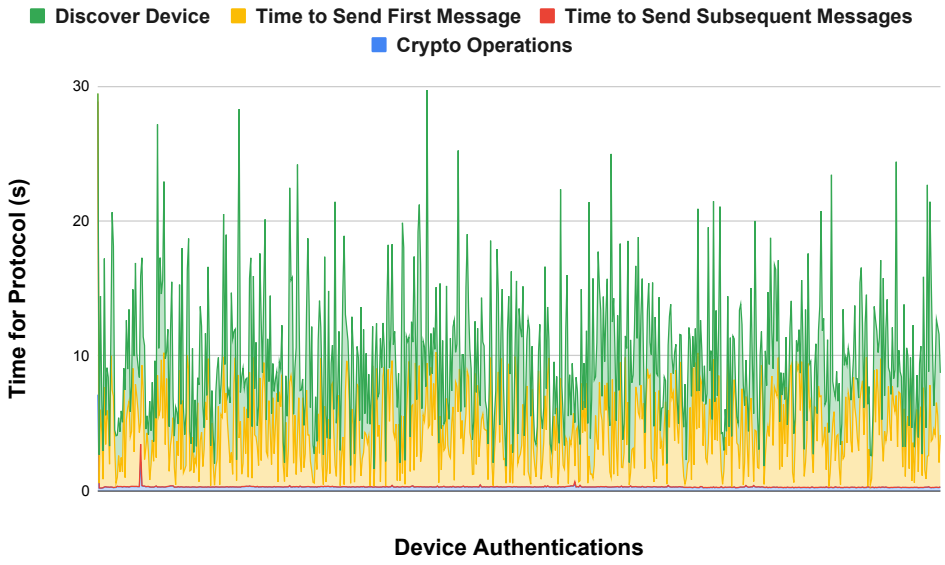

Figure 5.9: Breakdown of device authentication protocol steps.

linear increase in cost with the number of passwords.

We finally quantified the time for a user to add a new device using EDP. Unlike Bitwarden or other password managers which simply allow a user to log on to the password manager on any device, EDP requires the execution of its device authentication protocol. Figure 5.9 shows the execution time for 1000 instantiations of the device authentication protocol. The total time to complete the protocol ranged from 2 to 30 s, with an average of 10 s. Despite PAKEs being thought of as slow, the cryptographic operations take a negligible amount of time with all client-side steps of the PAKE protocol taking only 200 ms in total. The vast majority of the time is due to network latency from constructing the P2P connection, including device discovery through the topic and initializing the direct connection. Since we used real relay servers provided by the Hyperswarm developers, we experienced inconsistent delays, thus explaining the wide range of completion times; these delays can greatly improved with better infrastructure. The device authentication protocol represents a one-time cost per device, so even with an average wait time of 10 s, it is still a reasonable duration. While EDP imposes additional latency for this relatively rare setup operation, it provides the benefit of greater security for all operations.

5.7.2 Usability

Users' daily operation of EDP is no different from any other typical password manager. The primary difference that users experience in EDP is adding a new device by participating in the EDP device authentication protocol. Compared to a typical password manager, EDP additionally requires the user to copy a random phrase from an existing EDP device to the new one to setup the new device. Once setup is complete, the user can use the EDP client in the same manner as any other password manager client for regular functions like adding, requesting, and automatically inserting passwords. Since the key difference is in the setup process, we focused on evaluating the usability of EDP's new device setup with a small pilot user study.

Our IRB-approved user study (protocol number AAAT2330) included 20 users who installed and configured our EDP implementation on two devices. We allotted 60 minutes to the user study participants. Sixteen were 21 to 30 years old, three were 31 to 40 years old, and the last was 51 to 60 years old. Of these users, Six 21 to 30 year olds, one 31 to 40 year old, and the 51 to 60 year old subjects considered themselves to be non-technical users. All sessions were performed remotely due to the IRB prohibitions, so we asked users to install our EDP browser extension on two devices when possible; most participants installed EDP on a desktop computer and on a laptop computer. Users began the study by installing our extension, and then following a simple guide for getting started with EDP. The guide instructed them on choosing a username, master password, and providing these credentials to the EDP clients on their devices. Next, they initiated the process of adding a new device, copied the random phrase from one device to the other, and waited for the device authentication protocol to complete. All users finished the study within 5 to 10 minutes.

After completing the EDP device authentication protocol, all users completed the System Usability Scale (SUS) [61], an industry-standard survey for evaluating system usability. The SUS scores are summarized in Table 5.1, with a higher score indicating greater usability. Given the high results above 85 for nearly all participants, we can conclude that EDP's device authentication protocol is easy to understand and use [101]. We also expect that improvements to the UI such as graphical aids can clarify the process even further.

Mean	Stdev.	Min.	Q1	Median	Q3	Max
89	8	73	86	93	95	100

Table 5.1: System Usability Scale summarized scores.

5.8 Related Work

Password Managers. Commercial password managers encrypt individual passwords with a key derived from the user-provided master password [150, 151, 152, 144, 153]. This approach suffers from encryption keys with low entropy. The encrypted passwords stored on these password manager servers are susceptible to offline attacks if they are leaked; potential leaks of encrypted passwords stored on commercial services have previously occurred [130, 131]. Another concern is the possibility of fooling password managers into revealing passwords to adversaries via phishing or injection attacks [129]. Various approaches have tried to solve these problems. One defense against offline attacks is to avoid storing encrypted passwords [154, 144], for example by deterministically deriving high entropy passwords at runtime when they are needed. The basic cryptographic primitives for these works are key derivation functions (KDFs) such as PBKDF2 [155] and scrypt [156] among others. Other defenses against offline attacks include the use of decoy encrypted password files with plausible incorrect passwords to convert an attacker's work into impractical online attacks [157], and graphical password cues to remind users of their passwords rather than storing them [158]. KeePass [159] is an open source password manager that uses a master password, or a key file which is not necessarily a cryptographic key but any file the user wishes. These solutions however do not sufficiently address users with multiple devices. [157] has no mention of multiple devices, and [158] briefly mentions using a centralized server to synchronize user data but with no details on authenticating new devices. KeePass' key file abstraction does not support multiple keys, so it lacks a solution for multiple devices and granular revocation of them. Also, key files are arbitrary files chosen by users, and thus can be weak with low entropy and suffer from the same issue as weak master passwords.

Another approach is to use dual-possession authentication [160]. The basic design is to require

that a user with N trusted devices possess some subset of $K \leq N$ devices to be able to either retrieve or recreate their passwords. For example, password secret sharing schemes [161, 162, 163] split a given password into N parts across a user's devices and some share K of the parts are required to construct the correct password. Others store passwords on separate devices based on assigned security levels [164], or store encrypted passwords on one device which can only be decrypted by another device [160, 165]. A problem with dual-possession architectures is the assumption that users always have simultaneous access to at least two devices, e.g. a computer and smartphone, and that the password is only required on the computer, not the smartphone. In reality, users are increasingly authenticating to web services from their mobile devices while on the go with no access to a second device. Furthermore, such dual- or multi-possession designs are likely to be cumbersome to users, especially if they are authenticating to services which already use 2FA. For example, a user authenticating to Gmail with a dual-possession solution needs to confirm the password manager's password request with a secondary device, and then also complete Gmail's own 2FA step.

Apple's iCloud Keychain shares some attributes with EDP. Adding a device requires an existing device to allow the addition and a random number displayed on the existing device to be entered on the newly added device. While this usage model provides further evidence to support the usability of EDP's device authentication approach, the underlying mechanisms for iCloud Keychain are significantly different from EDP. iCloud Keychain requires users to trust Apple's servers and infrastructure [166], unlike EDP which does not require trusted infrastructure. iCloud Keychain allows users to decrypt their passwords with their low-entropy iCloud passwords and an iCloud Security Code (iCSC), which is a short 4 digit PIN. Although Apple claims that their back end infrastructure ensures that it is infeasible to brute force the iCSC, this again requires trusting Apple's word. iCloud's architecture and infrastructure are incredibly complex, requiring custom secure hardware and secured supply chains, all of which is claimed to be audited and inspected. For example, the Cloud Key Vault couples Apple's custom hardware security modules (HSMs) on users' devices with custom fleets of HSMs in iCloud. This amounts to a level of investment

and complexity that is inaccessible to the majority of password manager services that do not have access to capital like Apple. To add to the complexity, the iCloud and device HSMs utilize PKI maintained by Apple, such that the iCloud HSMs' CA certificate needs to be hardcoded into every Apple device. Finally, iCloud Keychain only runs on Apple hardware and operating systems. In contrast, EDP requires no trust in servers or the infrastructure, does not rely on weak master passwords for encryption, uses a simple architecture that can be used on any system, and is platform independent.

Key Directories and Management. EDP utilizes some concepts which have seen applications with similar motivations in previous work. CONIKS [167] describes a public key directory infrastructure to give users, rather than third-party monitors, agency in detecting and reporting malicious activity, including equivocation, by public key directory servers. Like CONIKS, EDP uses a prefix tree structure with privacy-preserving index keys, but the purpose and motivations differ. CONIKS instead uses a Merkle prefix tree to provide the necessary structure for key directories to publish signed tree roots, which are used by clients and auditors to detect equivocation; this is not necessary for EDP. CONIKS in general concerns itself with public keys which are exchanged among users for end-to-end encryption communications such as email. In contrast, EDP is designed for password management and never exchanges public keys with other users, only among an individual user's own devices.

5.9 Summary

We have designed Easy Device-based Passwords (EDP), a password manager system for improving the security of users' encrypted passwords at rest by using public key cryptography, without any changes to industry standard password manager app usage models. EDP encrypts users' passwords using self-generated keypairs on each of a user's devices, so we provide a PDK-based easy to use key management solution which uses a PAKE-based device authentication protocol to ensure mutual trust among a user's devices, without the user needing to know anything about public key cryptography and key management. Users perceive any key exchanges and authenti-

cation as a device synchronization and pairing step. Since EDP encrypts users' passwords with strong keys, the encrypted passwords can be stored on untrusted servers. We implemented EDP by retrofitting an existing password manager, showing that it requires only simple code changes for clients. Our experimental results show that EDP's password updates have reasonable overheads, and that the device authentication protocol is a quick and seamless experience for users.

Chapter 6: Conclusions and Future Work

Vectors for compromising private user data are not only limited to attacks using account credentials, but also include server-side and internal attacks that altogether bypass account authentication checks. Yet, the vast majority of online services act as if account credentials with 2FA are enough. In reality, user data is highly vulnerable. This situation can be greatly improved through the use of strong encryption, but encryption has traditionally been considered unusable for average users especially when multiple devices are involved, mainly due to the hurdles introduced by cryptographic key management [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]. The majority of users, even technical ones, first of all lack an understanding of the confusing and complex concepts and terminology used for cryptographic keys especially in the context of multiple device usage models, and this has been worsened by being coupled with crypto applications that have poor usability. To address these issues, I have designed, implemented, and evaluated a new key management philosophy which I call Per-Device Keys, or PDK for short, and also blueprints for applying PDK to enable encryption models which are transparent to users.

Although PDK is a key management solution, its unique design allows for an easy mapping of key management functions to equivalent device management ones. This then paves the path towards enabling strong, client-side encryption that is transparent to users. Transparent encryption processes are important because the actual steps of encrypting and decrypting data are confusing for users as well [13, 18]. Not only is there a need for usable key management, but also usable encryption. The exact properties of a usable encryption system are dependent on the context of the system and data being secured. Despite context-specific solutions, the general rule of thumb is to place encryption in the hands of user in a transparent fashion such that encryption and decryption occur automatically.

I have applied the PDK design in three different, representative contexts—email, photos, and

password managers—to solve their key management issues, while also developing new paradigms for strong, client-side encryption. E3 is a usable secure email solution which uses self-generated PDK keys with existing email encryption formats in a new way to achieve client-side encryption of emails on receipt. This approach protects all of a user's emails that are received prior to any attack, but is much easier to use than end-to-end encrypted email like PGP and S/MIME. Another important benefit of E3 is that it requires no server or protocol changes, so it can be easily deployed since users just need to install an E3 app to start using it. The experimental results for E3 show that it is compatible with existing IMAP services, has good performance, and is easy to understand and use.

ESP is a usable encrypted photos app which is compatible with existing services and has a transparent usage model that is not much different from normal photo apps that lack encryption. ESP introduces an image encryption technique which relies on encrypting JPEG images in a way that outputs valid JPEGs that are accepted by cloud photo services and are resilient to JPEG image compression. PDK addresses managing the self-generated keys used by ESP in a manner that is similar to E3, but with different restrictions such as needing to exchange public keys using the photo service itself. ESP requires no changes to existing cloud photo services, so users only need to install an ESP app to start using it. The experimental results show that ESP is robust against attacks, has acceptable performance overheads, and is simple to use.

EDP is a password manager that improves on the security of existing password managers without significant changes to their familiar and easy to understand usage models based on master passwords. EDP's new approach uses self-generated PDK keys to encrypt user passwords instead of weak, human-generated master passwords. The security of the device verification and enrollment portion of PDK is also improved in EDP so that it may be used with untrusted servers, the first password manager to allow this. The experimental results for EDP show that it is secure under these conditions with performance similar to existing password managers, and is usable.

6.1 Future Work

6.1.1 Integrating PDK with Services for Secure Key Storage

A feature not evaluated in this dissertation is the use of cloud key vaults (CKVs) such as Google Cloud Key Vault Service [168] and Apple's iCloud Keychain [166] to augment, but not replace, PDK's security and recovery guarantees. CKVs store users' secret keys which are protected by both the cloud service provider and user-memorized secrets such as passwords, PINs, and device lock screen patterns. Due to these multiple layers of protection, as well as special secure hardware such as HSMs on the CKV servers, the service provider themselves cannot access users' stored secret keys and recovery keys. Only users with knowledge or possession of multiple secrets may unlock access to their cryptographic keys stored in the cloud.

The drawback of these CKVs and the reason to not include them as part of PDK's design is the fact that using them would suggest an inherent reliance on a specific third party provider, which in turn introduces undesirable restrictions and constraints. For example, Google Cloud Key Vault Service only works on Android 9 Pie or newer, and Apple's iCloud Keychain is only compatible with Apple devices. In addition to this, they require special hardware and OS support. Both Google and Apple use custom secure hardware and servers to provide their security guarantees, as well as complex PKI schemes which also require support from server and user device OSes, so widespread support among consumer devices for CKVs is lacking. One of the goals which motivated PDK was a key management system which would be platform independent and work with any service, so the addition of CKVs would interfere with this.

But suppose that such secure hardware and PKI, both on servers and user devices, becomes the industry standard such that the vast majority of users' devices have the capability to leverage them. Then, PDK's security and recovery guarantees may be improved with the use of CKVs without detracting from the intended purpose of its design. The CKV architecture can improve the security of PDK by protecting PDK private keys using multiple layers of security provided by the hardware and HSMs. As noted in E3, ESP, and EDP, providing such hardware support is orthogonal to the

goals of these systems, but would indeed improve the security of PDK private keys stored on user devices as they effectively deter offline brute force attacks on encrypted private keys. For example, incorrecting guessing the lock screen PIN on Apple iPhones too many times will automatically wipe the device, thus rendering guessing attacks ineffective.

CKVs can also be used to securely store recovery keys to which the service providers do not have access. However, as discussed in Chapter 5 with respect to Apple iCloud Keychain, this assumes that these service providers are trustworthy, that their CKV architectures actually operate in the way which they are designed, and that their services do not become deprecated or discontinued in the future. If all of these conditions are satisfied, then the CKV service providers can improve the security of PDK since they would be unable to access user recovery keys while users would only need to provide a human-memorizable secret to gain access to them. The CKV service prevents compromises by enforcing a rate limit on attempts to access recovery keys through the use of its custom HSMs, meaning both the service and attackers cannot perform guessing attacks. This is a boon to PDK as it currently stands since its main recovery options are assuming users do not lose all their devices simultaneously, or that users have written down and safely stored their recovery keys where attackers cannot obtain them.

Future work could entail evaluating whether the steps involved with using a CKV together with PDK would provide a transparent and seamless experience to users. This has been shown to some degree with the widespread deployment and use of Apple iCloud Keychain, but only within the closed Apple ecosystem. It therefore remains to be seen whether other CKVs can be trusted and used independent of platform, for any service regardless of their lack of affiliation with Google and Apple.

6.1.2 Implications of Client-side Encryption for Services

Many cloud-based services provide their offerings at no cost to users. These free online services earn a profit not by charging users, but by collecting, using, and sometimes selling user data. One of the most notable examples in the industry is Google. Google provides a vast array of

useful, free services apart from Gmail and Google Photos which have already been discussed in this dissertation. Although many of Google's services do offer paid options, it is well known that Google's main income comes from its advertising and search platform which is driven by user data. Google services openly state that they collect information about their users and create astonishingly accurate profiles for them [169, 170], which helps Google to display advertisements relevant to users' interests. Since analyzing users and their data is so important to its business operations, Google is willing to provide its other services for free to build as large a userbase as possible. The work of this dissertation, namely strong client-side encryption with no reliance on servers, is therefore at odds with the business models of companies like Google, since they cannot analyze user data if it is encrypted by users.

This poses some potential issues for adopters of systems such as the ones described in this dissertation. In the worst case, companies with similar business models as Google may aggressively identify and ban users who are encrypting their data to discourage such practices. However, this kind of scorched-earth policy seems unlikely, at least for now. We can look at existing cases to approximate how Google may respond; for example, the case of ad blockers. Ad blockers, which often come in the form of web browser extensions such as AdBlock [171], Adblock Plus [172], and uBlock Origin [173], are installed by users to hide advertisements from being shown to them on the web, and thus are completely at odds with Google's advertising business model. Despite this, Google has not banned these ad blockers; these extensions are in fact prominently displayed as among the most popular extensions on the Google Chrome web browser's extension store.

Google may have a lenient attitude towards web browser ad blockers for at least a few reasons, some of which can be extrapolated to the case of client-side encryption. The first is that the number of ad blockers users, despite each of the three mentioned ad blockers having at least over 10 million downloads, likely represent only a small fraction of the total number of users on the web. Furthermore, ad blocker users are unlikely to pay attention to advertisements at all even when shown them. Google may therefore feel that the loss of potential revenue from ad blocker users is unavoidable, and that punishing them by banning ad blocker extensions would yield no

benefit regardless. Systems for client-side encryption such as E3 and ESP may fall under the same treatment as users of ad blockers unless adoption of client-side encryption drastically eclipses the number of ad blocker users.

However, another possible explanation for Google's lax policy towards web browser ad blockers is the increasing importance of mobile device users on the web, and Google may have decided to focus its efforts on this market. Android users generally cannot use ad blockers because Google specifically bans them from the Android platform and the Google Play Store; Android web browsers consequently lack any ad blocking features. Thus, if user adoption rates of client-side encryption outstrips that of users who do not use encryption, there is precedent for Google possibly opting to ban the use of client-side encryption with its services, which would mark a death knell for the systems described in this dissertation.

On the bright side, it may be possible through either system design or cooperation with services to avoid this outcome. For example, as discussed in Chapter 3, E3 is compatible with spam filters because it only encrypts email after it has been received, giving the email service time to scan and filter emails; similarly, this gives time for services such as Google to analyze and anonymize user data gleaned from emails. Thus, E3's encrypt on receipt design is unlikely to move Google to punitive measures even if it becomes widely used.

Unlike E3, ESP poses more of a challenge due to its design in which users encrypt their photos before uploading them to services such as Google Photos. Assuming that the threat is a future compromise of users' photo hosting service accounts, a naive solution, similar to E3's model, is to have ESP clients upload user photos unencrypted, giving the photo hosting service the opportunity to scan the image, before replacing it with an encrypted copy. This would however severely impact performance, as then users would need to upload even more data per image. Other schemes may be possible, such as using homomorphic encryption—which Google is exploring for exchanging data on the business side with other companies [174]—to allow Google to compute results from user data without necessarily having access to the raw data itself. More generally, a system design for client-side encryption that allows services such as Google to learn anonymized results about

user data may satisfy their requirements; but exploration of these kinds of approaches are left as future work.

As with all secure systems, the key question is, "What is the threat model?" As a simple example, if we relax the threat model for ESP, then it may be acceptable to users if only their most sensitive of photos are encrypted, while the less important ones are left in the clear; this may be enough to satisfy Google. And the proposed solution of uploading an unencrypted photo to give the service the opportunity to analyze it, then replacing it with an encrypted version assumes that the main threat is a future attacker who compromises users' photo hosting service accounts. It does not assume that the service itself is the threat to protect against, whether it is honest-but-curious or outright malicious.

The likely outcome is that, similar to how this dissertation required drastically different system solutions for email, photo hosting, and password management, different services will require different relaxing of users' threat models in order to develop client-side encryption approaches which do not incite aggressive countermeasures by services. But at the same time, even Google has mostly ignored web browser extension ad blockers, only taking action against them on Android, suggesting that services such as Google will only take action at certain breaking points. For now, client-side encryption approaches with existing services is a largely untapped idea, so these questions are left for future work if adoption rates soar, perhaps in the event that services like Google finally do decide to take action.

References

[1] B. Ives, K. R. Walsh, and H. Schneider, "The Domino Effect of Password Reuse," *Communications of the ACM*, vol. 47, no. 4, 75–78, Apr. 2004.

[2] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang, "The Tangled Web of Password Reuse," in *26th Annual Network and Distributed System Security Symposium (NDSS 2014)*, vol. 14, 2014, pp. 23–26.

[3] A. Visconti, S. Bossi, H. Ragab, and A. Calò, "On the Weaknesses of PBKDF2," in *Cryptology and Network Security*, 2015, pp. 119–126.

[4] A. Ruddick and J. Yan, "Acceleration Attacks on PBKDF2: Or, What Is Inside the Black-Box of oclHashcat?" In *10th USENIX Workshop on Offensive Technologies (WOOT 2016)*, Austin, Texas: USENIX Association, Aug. 2016.

[5] W. Palant, *Is your LastPass data really safe in the encrypted online vault?* `https://palant.de/2018/07/09/is-your-lastpass-data-really-safe-in-the-encrypted-online-vault/`, Jul. 2018.

[6] J. Cox and M. Hoppenstedt, *Sources: Facebook Has Fired Multiple Employees for Snooping on Users - VICE*, `https://www.vice.com/en_us/article/bjp9zv/facebook-employees-look-at-user-data`, May 2018.

[7] J. Cox, *Snapchat Employees Abused Data Access to Spy on Users - VICE*, `https://www.vice.com/en_uk/article/xwnva7/snapchat-employees-abused-data-access-spy-on-users-snaplion`, May 2019.

[8] B. News, *Trend Micro rogue employee exposes customer data - BBC News*, `https://www.bbc.com/news/technology-50315544`, Nov. 2019.

[9] J. Cox, *Ring Fired Employees for Watching Customer Videos - VICE*, `https://www.vice.com/en_us/article/y3mdvk/ring-fired-employees-abusing-video-data`, Jan. 2020.

[10] Google, Inc., *Google Terms of Service – Privacy & Terms – Google*, `https://policies.google.com/terms`, Jan. 2020.

[11] SmugMug and Flickr, *Help | Flickr*, `https://www.flickr.com/help/privacy`, Jan. 2020.

[12] D. Moriyama and M. Yung, "The Bright Side Arguments for the Coming Smartphones Crypto War: The Added Value of Device Encryption," in *2015 IEEE Conference on Communications and Network Security (CNS 2015)*, Florence, Italy, Sep. 2015, pp. 65–73.

[13] A. Whitten and J. D. Tygar, "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0," in *Proceedings of the 8th USENIX Security Symposium (USENIX Security 1999)*, Washington, D.C.: USENIX Association, Aug. 1999, pp. 169–184.

[14] S. L. Garfinkel and R. C. Miller, "Johnny 2: A User Test of Key Continuity Management with S/MIME and Outlook Express," in *Proceedings of the 2005 Symposium on Usable Privacy and Security (SOUPS 2005)*, Pittsburgh, Pennsylvania: Association for Computing Machinery, Jul. 2005, 13–24, ISBN: 1595931783.

[15] S. Sheng, L. Broderick, C. A. Koranda, and J. J. Hyland, "Why Johnny Still Can't Encrypt: Evaluating the Usability of Email Encryption Software," in *Proceedings of the 2006 Symposium on Usable Privacy and Security - Poster Session (SOUPS 2006)*, Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, Jul. 2006, 13–24.

[16] S. Clark, T. Goodspeed, P. Metzger, Z. Wasserman, K. Xu, and M. Blaze, "Why (Special Agent) Johnny (Still) Can't Encrypt: A Security Analysis of the APCO Project 25 Two-Way Radio System," in *Proceedings of the 20th USENIX Conference on Security (USENIX Security 2011)*, USENIX Association, 2011.

[17] S. Fahl, M. Harbach, T. Muders, M. Smith, and U. Sander, "Helping Johnny 2.0 to Encrypt His Facebook Conversations," in *Proceedings of the 2012 Symposium on Usable Privacy and Security (SOUPS 2012)*, Washington, D.C.: Association for Computing Machinery, 2012, ISBN: 9781450315326.

[18] S. Ruoti, N. Kim, B. Burgon, T. Van Der Horst, and K. Seamons, "Confused Johnny: When Automatic Encryption Leads to Confusion and Mistakes," in *Proceedings of the 9th Symposium on Usable Privacy and Security (SOUPS 2013)*, Association for Computing Machinery, Newcastle, United Kingdom: Association for Computing Machinery, Jul. 2013.

[19] Z. Benenson, G. Lenzini, D. Oliveira, S. Parkin, and S. Uebelacker, "Maybe Poor Johnny Really Cannot Encrypt: The Case for a Complexity Theory for Usable Security," in *Proceedings of the 2015 New Security Paradigms Workshop (NSPW 2015)*, Twente, Netherlands: Association for Computing Machinery, 2015, 85–99, ISBN: 9781450337540.

[20] E. Atwater, C. Bocovich, U. Hengartner, E. Lank, and I. Goldberg, "Leading Johnny to Water: Designing for Usability and Trust," in *Proceedings of the 2015 Symposium On Usable Privacy and Security (SOUPS 2015)*, Ottawa, Canada: USENIX Association, Jul. 2015, pp. 69–88, ISBN: 978-1-931971-249.

[21] H. Orman, "Why Won't Johnny Encrypt?" *IEEE Internet Computing*, vol. 19, no. 1, pp. 90–94, Jan. 2015.

[22] S. Ruoti, J. Andersen, D. Zappala, and K. E. Seamons, "Why Johnny Still, Still Can't Encrypt: Evaluating the Usability of a Modern PGP Client," *Computing Research Repository (CoRR)*, vol. abs/1510.08555, Jan. 2016. arXiv: 1510.08555.

[23] S. Ruoti, J. Andersen, S. Heidbrink, M. O'Neill, E. Vaziripour, J. Wu, D. Zappala, and K. Seamons, ""We're on the Same Page": A Usability Study of Secure Email Using Pairs of Novice Users," in *Proceedings of the 2016 Conference on Human Factors in Computing Systems (CHI 2016)*, San Jose, California, USA: Association for Computing Machinery, May 2016, pp. 4298–4308, ISBN: 978-1-4503-3362-7.

[24] A. Herzberg and H. Leibowitz, "Can Johnny Finally Encrypt? Evaluating E2E-Encryption in Popular IM Applications," in *Proceedings of the 6th Workshop on Socio-Technical Aspects in Security and Trust (STAST 2016)*, Los Angeles, California: Association for Computing Machinery, 2016, pp. 17–28, ISBN: 9781450348263.

[25] Apple, Inc., *Use FileVault to encrypt the startup disk on your Mac - Apple Support*, https://support.apple.com/en-us/HT204837, Sep. 2020.

[26] Microsoft, Inc., *Finding your BitLocker recovery key in Windows 10*, https://support.microsoft.com/en-us/help/4530477/windows-10-finding-your-bitlocker-recovery-key, Jan. 2020.

[27] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano, "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes," in *2012 IEEE Symposium on Security and Privacy (S&P 2012)*, San Francisco, California, May 2012, pp. 553–567.

[28] T. Hunt, *Have I Been Pwned: Check if your email has been compromised in a data breach*, https://haveibeenpwned.com/, Jan. 2021.

[29] J. S. Koh, S. M. Bellovin, and J. Nieh, "Why Joanie Can Encrypt: Easy Email Encryption with Easy Key Management," in *Proceedings of the 14th European Conference on Computer Systems (EuroSys 2019)*, Dresden, Germany, Mar. 2019, 2:1–2:16.

[30] J. S. Koh, J. Nieh, and S. M. Bellovin, "Encrypted Cloud Photo Storage Using Google Photos," in *Proceedings of the 19th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys 2021)*, Mars, Solar System, Milky Way: Association for Computing Machinery, Jun. 2021–Jul. 2021.

[31] G. Krieg and T. Kopan, *Is this the email that hacked John Podesta's account?* CNN Politics, Oct. 2016.

[32] R. Windrem, *Payback? Russia Gets Hacked, Revealing Putin Aide's Secrets*, NBC News, Oct. 2016.

[33] The Washington Times, *Hacker wanted to 'derail' Palin*, Sep. 2008.

[34] L. Franceschi-Bicchierai, *Teen hackers: A '5-year-old' could have hacked into CIA director's emails*, VICE Motherboard, Oct. 2015.

[35] R. S. Jeff Donn Desmond Butler, *Russian hackers hunt hi-tech secrets, exploiting US weakness*, Associated Press, Feb. 2018.

[36] Z. Whittaker, *Servers of email host used in US school bomb threats seized by German police*, Dec. 2015.

[37] K. Poulsen, *If You Used This Secure Webmail Site, the FBI Has Your Inbox*, WIRED, Jan. 2014.

[38] WikiLeaks, *WikiLeaks — Sony Archives*, WikiLeaks, 2018.

[39] Google, Inc., *Google One - More storage and extra benefits from Google*, `https://one.google.com/about`, Feb. 2019.

[40] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "OpenPGP Message Format," IETF, RFC 4880, Nov. 2007, p. 89.

[41] J. Müller, M. Brinkmann, D. Poddebniak, S. Schinzel, and J. Schwenk, "Mailto: Me Your Secrets. On Bugs and Features in Email End-to-End Encryption," in *2020 IEEE Conference on Communications and Network Security (CNS 2020)*, Jun. 2020, pp. 1–9.

[42] S. L. Garfinkel and A. Shelat, "Remembrance of Data Passed: A Study of Disk Sanitization Practices," *IEEE Security & Privacy*, vol. 1, no. 1, pp. 17–27, Jan. 2003–Feb. 2003.

[43] B. Specht, *Email Client Market Share Trends for the First Half of 2018 – Litmus Software, Inc.* `https://litmus.com/blog/email-client-market-share-trends-first-half-of-2018`, Jul. 2018.

[44] M. R. Crispin, "INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1," IETF, RFC 3501, Mar. 2003, p. 108.

[45] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," IETF, RFC 5280, May 2008, p. 151.

[46] B. Ramsdell and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification," IETF, RFC 5751, Jan. 2010, p. 45.

[47] S. Brandt, "IMAP REPLACE extension," IETF, RFC 7162, Jan. 2019, p. 11.

[48] C. Newman, "Using TLS with IMAP, POP3 and ACAP," IETF, RFC 2595, Jun. 1999, p. 15.

[49] A. Melnikov and D. Cridland, "IMAP Extensions: Quick Flag Changes Resynchronization (CONDSTORE) and Quick Mailbox Resynchronization (QRESYNC)," IETF, RFC 7162, May 2014, p. 52.

[50] A. J. Aviv, M. E. Locasto, S. Potter, and A. D. Keromytis, "SSARES: Secure Searchable Automated Remote Email Storage," in *23rd Annual Computer Security Applications Conference, 2007 (ACSAC 2007)*, IEEE, Miami Beach, Florida, USA: ACSA, Dec. 2007, pp. 129–139.

[51] Google, Inc., *roughtime - Git at Google*, `https://roughtime.googlesource.com/roughtime`, 2018.

[52] P. Juola and P. Zimmermann, "Whole-Word Phonetic Distances and the PGPfone Alphabet," in *Proceeding of 4th International Conference on Spoken Language Processing (ICSLP 1996)*, vol. 1, Philadelphia, PA, USA: IEEE, Oct. 1996, pp. 98–101.

[53] M. Farb, Y.-H. Lin, T. H.-J. Kim, J. McCune, and A. Perrig, "Safeslinger: Easy-to-use and secure public-key exchange," in *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking (MobiCom 2013)*, Miami, Florida, USA: Association for Computing Machinery, 2013, pp. 417–428, ISBN: 978-1-4503-1999-7.

[54] Autocrypt Team, *Autocrypt 1.0.1 documentation*, `https://autocrypt.org/`, 2016.

[55] Google, Inc., *How Google retains data we collect – Privacy & Terms – Google*, `https://policies.google.com/technologies/retention?hl=en`, 2019.

[56] US Federal Trade Commission, *Federal Trade Commission Act - Section 5: Unfair or Deceptive Acts or Practices*, `https://www.federalreserve.gov/boarddocs/supmanual/cch/ftca.pdf`, Dec. 2016.

[57] ——, *15 USC CHAPTER 2, SUBCHAPTER I: FEDERAL TRADE COMMISSION*, `http://uscode.house.gov/view.xhtml?req=granuleid%3AUSC-prelim-title15-chapter2-subchapter1&edition=prelim`, Sep. 1914.

[58] rtyley, *Spongycastle by rtyley*, Github.io, 2018.

[59] Apple, Inc., *SecACLCreateWithSimpleContents - Security | Apple Developer Documentation*, `https://developer.apple.com/documentation/security/1402295-secaclcreatewithsimplecontents?language=objc`, 2018.

[60] Google, Inc., *KeyChain | Android Developers*, https://developer.android.com/reference/android/security/KeyChain, 2018.

[61] U.S. Department of Health & Human Services, *System Usability Scale (SUS)*, 2015.

[62] S. Ruoti, J. Andersen, T. Hendershot, D. Zappala, and K. Seamons, "Private Webmail 2.0: Simple and Easy-to-Use Secure Email," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST 2016)*, Tokyo, Japan: Association for Computing Machinery, Oct. 2016, pp. 461–472, ISBN: 978-1-4503-4189-9.

[63] S. L. Garfinkel, "Enabling Email Confidentiality Through the Use of Opportunistic Encryption," in *Proceedings of the 2003 Annual National Conference on Digital Government Research (dg.o 2003)*, Boston, Massachusetts: Digital Government Society of North America, May 2003, pp. 1–4.

[64] W. Koch and M. Brinkmann, "STEED–Usable End-to-End Encryption," White Paper, Oct. 2011.

[65] A. Lerner, E. Zeng, and F. Roesner, "Confidante: Usable Encrypted Email: A Case Study with Lawyers and Journalists," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, Paris, France: IEEE, Apr. 2017, pp. 385–400.

[66] Lavabit, LLC., *Lavabit*, https://lavabit.com/, 2004.

[67] Posteo, *Email green, secure, simple and ad-free - posteo.de - Features*, https://posteo.de/en/site/features, 2009.

[68] Tutanota, *Secure email: Tutanota makes encrypted emails easy.* https://tutanota.com/, 2011.

[69] M. Cardwell, *Automatically Encrypting all Incoming Email*, https://www.grepular.com/Automatically_Encrypting_all_Incoming_Emails, Blog, 2011.

[70] B. Jacke, *How to automatically PGP/MIME encrypt incoming mail via procmail*, https://www.j3e.de/pgp-mime-encrypt-in-procmail.html, Oct. 2008.

[71] Posteo, *Help - How do I activate inbound encryption with my public PGP key? - posteo.de*, https://posteo.de/en/help/how-do-i-activate-inbound-encryption-with-my-public-pgp-key, 2015.

[72] S. Heyman, *Photos, Photos Everywhere - The New York Times*, https://www.nytimes.com/2015/07/23/arts/international/photos-photos-everywhere.html, Jul. 2015.

[73] H. McCracken, *How Google Photos reached a billion users*, https://www.fastcompany.com/90380618/how-google-photos-joined-the-billion-user-club, Jul. 2019.

[74] J. Kastrenakes, *Apple denies iCloud breach in celebrity nude photo hack - The Verge*, https://www.theverge.com/2014/9/2/6098107/apple-denies-icloud-breach-celebrity-nude-photo-hack, Sep. 2014.

[75] L. Abrams, *Google Bug Sent Private Google Photos Videos to Other Users*, https://www.bleepingcomputer.com/news/google/google-bug-sent-private-google-photos-videos-to-other-users/, Feb. 2020.

[76] M. Zhu, T. Moataz, and S. Kamara, *Pixek*, https://pixek.io/, Nov. 2019.

[77] P3 Privacy, *P3 image*, https://www.p3image.com/, Nov. 2019.

[78] M.-R. Ra, R. Govindan, and A. Ortega, "P3: Toward Privacy-preserving Photo Sharing," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI 2013)*, Lombard, Illinois: USENIX Association, Apr. 2013, pp. 515–528.

[79] L. Yuan, D. McNally, A. Küpçü, and T. Ebrahimi, "Privacy-preserving photo sharing based on a public key infrastructure," in *Applications of Digital Image Processing XXXVIII*, A. G. Tescher, Ed., International Society for Optics and Photonics, vol. 9599, SPIE, Sep. 2015, pp. 515 –527.

[80] L. Yuan, P. Korshunov, and T. Ebrahimi, "Privacy-Preserving Photo Sharing based on a Secure JPEG," in *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, vol. 2015, Apr. 2015.

[81] L. Zhang, T. Jung, C. Liu, X. Ding, X.-Y. Li, and Y. Liu, "POP: Privacy-Preserving Outsourced Photo Sharing and Searching for Mobile Devices," in *2015 IEEE 35th International Conference on Distributed Computing Systems*, Jun. 2015, pp. 308–317.

[82] J. He, B. Liu, D. Kong, X. Bao, N. Wang, H. Jin, and G. Kesidis, "PUPPIES: Transformation-Supported Personalized Privacy Preserving Partial Image Sharing," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2016)*, Jun. 2016, pp. 359–370.

[83] R. A. Fisher and F. Yates, *Statistical Tables: For Biological, Agricultural and Medical Research*. Oliver and Boyd, 1938.

[84] J. Alakuijala, R. Obryk, O. Stoliarchuk, Z. Szabadka, L. Vandevenne, and J. Wassenberg, "Guetzli: Perceptually Guided JPEG Encoder," *Computing Research Repository (CoRR)*, vol. abs/1703.04421, Mar. 2017. arXiv: 1703.04421.

[85] ——, *google/guetzli: Perceptual JPEG encoder*, https://github.com/google/guetzli, Jan. 2020.

[86] Google, Inc., *Download Open Images V5*, https://storage.googleapis.com/openimages/web/download.html, Jan. 2020.

[87] Apple, Inc., *Use FileVault to encrypt the startup disk on your Mac - Apple Support*, https://support.apple.com/en-us/HT204837, Jan. 2020.

[88] ——, *Photos - Private, on-device technologies to browse and edit photos and videos on iOS and iPadOS*, https://www.apple.com/ios/photos/pdf/Photos_Tech_Brief_Sept_2019.pdf, Sep. 2019.

[89] Google, Inc., *ML Kit | Google Developers*, https://developers.google.com/ml-kit, Jan. 2020.

[90] Apple Inc., *Core ML - Machine Learning - Apple Developer*, https://developer.apple.com/machine-learning/core-ml/, Jan. 2020.

[91] M. Abdalla and D. Pointcheval, "Simple Password-Based Encrypted Key Exchange Protocols," in *Proceedings of the 2005 International Conference on Topics in Cryptology (CT-RSA 2005)*, San Francisco, California: Springer-Verlag, 2005, pp. 191–208, ISBN: 3540243992.

[92] T. Chuman, K. Kurihara, and H. Kiya, "On the Security of Block Scrambling-Based EtC Systems against Extended Jigsaw Puzzle Solver Attacks," *IEICE Transactions on Information and Systems*, vol. 101, no. 1, pp. 37–44, Jan. 2018.

[93] T. Chuman, W. Sirichotedumrong, and H. Kiya, "Encryption-Then-Compression Systems Using Grayscale-Based Image Encryption for JPEG Images," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1515–1525, Jun. 2019.

[94] Y. Wu, J. P. Noonan, and S. Agaian, "NPCR and UACI Randomness Tests for Image Encryption," *Cyber Journals: Multidisciplinary Journals in Science and Technology, Journal of Selected Areas in Telecommunications (JSAT)*, vol. 1, no. 2, pp. 31–38, Apr. 2011.

[95] G. Chen, Y. Mao, and C. K. Chui, "A symmetric image encryption scheme based on 3D chaotic cat maps," *Chaos, Solitons & Fractals*, vol. 21, no. 3, pp. 749–761, Jul. 2004.

[96] SimpleMobileTools, *SimpleMobileTools/Simple-Gallery: Browse your memories without any interruptions with this photo and video gallery*, https://github.com/SimpleMobileTools/Simple-Gallery, Jan. 2020.

[97] Facebook Open Source, *Fresco - An image management library. | Fresco*, https://frescolib.org/, Jan. 2020.

[98] libjpeg-turbo, *libjpeg-turbo | Main / libjpeg-turbo*, https://libjpeg-turbo.org/, Jan. 2020.

[99] Free Software Foundation, *The GNU MP Bignum Library*, `https://gmplib.org/`, Jan. 2020.

[100] I. T. Union, "Objective perceptual multimedia video quality measurement in the presence of a full reference," International Telecommunication Union, Geneva, CH, Recommendation, Aug. 2008.

[101] J. M. Aaron Bangor Philip Kortum, "Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale," in *Journal of Usability Studies*, JUS, vol. 4, May 2009, pp. 114–123.

[102] J. Fridrich, "Image Encryption Based On Chaotic Maps," in *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, vol. 2, Oct. 1997, pp. 1105–1110.

[103] Z.-l. Zhu, W. Zhang, K.-w. Wong, and H. Yu, "A Chaos-Based Symmetric Image Encryption Scheme Using a Bit-Level Permutation," *Inf. Sci.*, vol. 181, no. 6, pp. 1171–1186, Mar. 2011.

[104] Y.-Q. Zhang and X. Wang, "Analysis and Improvement of a Chaos-based Symmetric Image Encryption Scheme Using a Bit-level Permutation," *Nonlinear Dynamics*, vol. 77, pp. 687–698, Aug. 2014.

[105] C. Fu, J.-B. Huang, N.-N. Wang, Q.-B. Hou, and W. Lei, "A Symmetric Chaos-Based Image Cipher with an Improved Bit-Level Permutation Strategy," *Entropy*, vol. 16, Jan. 2014.

[106] W. Xingyuan and Z. Hongyu, "Cracking and Improvement of an Image Encryption Algorithm Based on Bit-Level Permutation and Chaotic System," *IEEE Access*, vol. 7, pp. 112 836–112 847, Aug. 2019.

[107] Z. Hua, Y. Zhou, and H. Huang, "Cosine-transform-based Chaotic System for Image Encryption," *Information Sciences*, vol. 480, pp. 403 –419, Apr. 2019.

[108] J. Black and P. Rogaway, "Ciphers with Arbitrary Finite Domains," in *Cryptographers' Track at the RSA Conference*, Springer, Feb. 2002, pp. 114–130.

[109] Hongjun Wu and Di Ma, "Efficient and secure encryption schemes for JPEG2000," in *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, May 2004, pp. V–869.

[110] M. Tierney, I. Spiro, C. Bregler, and L. Subramanian, "Cryptagram: Photo Privacy for Online Social Media," in *COSN 2013 - Proceedings of the 2013 Conference on Online Social Networks*, Association for Computing Machinery, Oct. 2013, pp. 75–87, ISBN: 9781450320849.

[111] F. Dufaux and T. Ebrahimi, "Toward a secure JPEG," in *Applications of Digital Image Processing XXIX*. Society of Photo-Optical Instrumentation Engineers (SPIE), Sep. 2006, vol. 6312, 63120K.

[112] K. Wong and K. Tanaka, "DCT based scalable scrambling method with reversible data hiding functionality," in *2010 4th International Symposium on Communications, Control and Signal Processing (ISCCSP)*, Mar. 2010, pp. 1–4.

[113] T. Honda, Y. Murakami, Y. Yanagihara, T. Kumaki, and T. Fujino, "Hierarchical Image-scrambling Method with Scramble-level Controllability for Privacy Protection," *2013 IEEE 56th International Midwest Symposium on Circuits and Systems (MWSCAS 2013)*, pp. 1371–1374, Aug. 2013.

[114] L. Yuan, P. Korshunov, and T. Ebrahimi, "Secure JPEG Scrambling Enabling Privacy in Photo Sharing," in *11th IEEE International Conference and Workshops on Automatic Face and Gesture Recognition (FG 2015)*, vol. 04, May 2015, pp. 1–6.

[115] X. Niu, C. Zhou, J. Ding, and B. Yang, "JPEG Encryption with File Size Preservation," in *2008 International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, Aug. 2008, pp. 308–311.

[116] K. Minemura, Z. Moayed, K. Wong, X. Qi, and K. Tanaka, "JPEG Image Scrambling Without Expansion in Bitstream Size," in *2012 19th IEEE International Conference on Image Processing*, Sep. 2012, pp. 261–264.

[117] J. He, S. Huang, S. Tang, and J. Huang, "JPEG Image Encryption With Improved Format Compatibility and File Size Preservation," *IEEE Transactions on Multimedia*, vol. 20, no. 10, pp. 2645–2658, Oct. 2018.

[118] C. V. Wright, W.-c. Feng, and F. Liu, "Thumbnail-Preserving Encryption for JPEG," in *Proceedings of the 3rd ACM Workshop on Information Hiding and Multimedia Security*, ser. IH & MMSec '15, Portland, Oregon, USA: ACM, Jun. 2015, pp. 141–146, ISBN: 978-1-4503-3587-4.

[119] K. Tajik, A. Gunasekaran, R. Dutta, B. Ellis, R. B. Bobba, M. Rosulek, C. V. Wright, and W. Feng, "Balancing Image Privacy and Usability with Thumbnail-Preserving Encryption," in *26th Annual Network and Distributed System Security Symposium, (NDSS 2019)*, San Diego, California, Feb. 2019.

[120] S. Shan, E. Wenger, J. Zhang, H. Li, H. Zheng, and B. Y. Zhao, "Fawkes: Protecting Privacy against Unauthorized Deep Learning Models," in *29th USENIX Security Symposium (USENIX Security 2020)*, USENIX Association, Aug. 2020, pp. 1589–1604, ISBN: 978-1-939133-17-5.

[121] K. Hill, *This Tool Could Protect Your Photos From Facial Recognition - The New York Times*, https://www.nytimes.com/2020/08/03/technology/fawkes-tool-protects-photos-from-facial-recognition.html?action=click&module=News&pgtype=Homepage, Aug. 2020.

[122] J. D. Biersdorfer, *Taking Your Passwords With You Anywhere - The New York Times*, https://www.nytimes.com/2017/07/07/technology/personaltech/password-manager-apps.html, Jul. 2017.

[123] G. A. Fowler, *Are password managers safe? A new report finds flaws in five. - The Washington Post*, https://www.washingtonpost.com/, Feb. 2019.

[124] Best Reviews, *The Top Password Managers According to Security Experts and Researchers - Best Reviews*, https://password-managers.bestreviews.net/the-top-password-managers-according-to-security-experts-and-researchers/, Apr. 2020.

[125] National Cyber Security Centre, *Password manager buyers guide - NCSC.GOV.UK*, https://www.ncsc.gov.uk/collection/passwords/password-manager-buyers-guide, Apr. 2020.

[126] A. Huth, M. Orlando, and L. Pesante, *Password Security, Protection, and Management*, http://aahuth.com/wp-content/uploads/sites/44/2014/02/PasswordMgmt2012-2.pdf, 2012.

[127] Independent Security Evaluators, *Password Managers: Under the Hood of Secrets Management - Independent Security Evaluators*, https://www.ise.io/casestudies/password-manager-hacking/, Feb. 2019.

[128] J. Blocki, B. Harsha, and S. Zhou, "On the Economics of Offline Password Cracking," in *2018 IEEE Symposium on Security and Privacy (S&P 2018)*, 2018, pp. 853–871.

[129] Z. Li, W. He, D. Akhawe, and D. Song, "The Emperor's New Password Manager: Security Analysis of Web-based Password Managers," in *23rd USENIX Security Symposium (USENIX Security 2014)*, San Diego, California: USENIX Association, Aug. 2014, pp. 465–479, ISBN: 978-1-931971-15-7.

[130] J. Raphael, *LastPass CEO Explains Possible Hack | PCWorld*, https://www.pcworld.com/article/227268/lastpass_ceo_exclusive_interview.html, May 2011.

[131] Z. Whittaker, *Password manager OneLogin hacked, exposing sensitive customer data | ZDNet*, https://www.zdnet.com/article/onelogin-hit-by-data-breached-exposing-sensitive-customer-data/, Jun. 2017.

[132] SecLists.org, *Full Disclosure: Keeper Commander*, https://seclists.org/fulldisclosure/2018/May/41, May 2018.

[133] bitwarden, *bitwarden/browser: The browser extension vault (Chrome, Firefox, Opera, Edge, Safari, & more)*. https://github.com/bitwarden/browser, Apr. 2020.

[134] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, Feb. 1997.

[135] T. M. A. Lomas, L. Gong, J. H. Saltzer, and R. M. Needham, "Reducing Risks from Poorly Chosen Keys," in *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP 1989)*, SIGOPS, Dec. 1989, pp. 14–18.

[136] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976.

[137] G. Hazel and A. Norberg, *BEP-9: Extension for Peers to Send Metadata Files*, http://bittorrent.org/beps/bep_0009.html, Mar. 2017.

[138] amark, *Gun*, https://github.com/amark/gun, Apr. 2020.

[139] A. C. Mark Nadal, *A Trustless Decentralized Bandwidth Incentive*, https://web.stanford.edu/~nadal/A-Decentralized-Data-Synchronization-Protocol.pdf, Nov. 2017.

[140] ——, *A Decentralized Data Synchronization Protocol*, https://web.stanford.edu/~nadal/A-Decentralized-Data-Synchronization-Protocol.pdf, Nov. 2017.

[141] mafintosh, *mafintosh/append-tree: Model a tree structure on top off an append-only log*. https://github.com/mafintosh/append-tree, Apr. 2020.

[142] OrbitDB Community, *Home – OrbitDB*, https://orbitdb.org/, Apr. 2020.

[143] J. Benet, *IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3)*, https://github.com/ipfs/ipfs/blob/master/papers/ipfs-cap2pfs/ipfs-p2p-file-system.pdf, Jul. 2014.

[144] S. Huber and S. Rasthofer, *Infocondb*, https://infocondb.org/con/def-con/def-con-25/bypassing-android-password-manager-apps-without-root, Jul. 2017.

[145] S. Helme, *The Impending Doom of Expiring Root CAs and Legacy Clients*, Jun. 8, 2020.

[146] mafintosh, *hyperswarm/hyperswarm: A distributed networking stack for connecting peers.* https://github.com/hyperswarm/hyperswarm, Apr. 2020.

[147] D. P. Jablon, "Strong Password-Only Authenticated Key Exchange," *SIGCOMM Comput. Commun. Rev.*, vol. 26, no. 5, pp. 5–26, Oct. 1996.

[148] IEEE, *IEEE Standard Specification for Password-Based Public-Key Cryptographic Techniques*, Jan. 2009.

[149] ISO, *Information technology — Security techniques — Key management — Part 4: Mechanisms based on weak secrets*, Nov. 2017.

[150] LogMeIn, Inc., *How It Works*, https://www.lastpass.com/how-lastpass-works, Apr. 2020.

[151] 1Password, *Keep your secrets & passwords safe and secure | 1Password*, https://1password.com/security/, Apr. 2020.

[152] Dashlane, *Pricing, Premium, and Free Trial Plans | Dashlane*, https://www.dashlane.com/plans, Apr. 2020.

[153] Bitwarden Inc., *Open Source Password Manager for Individuals and Teams | Bitwarden*, https://bitwarden.com/, Apr. 2020.

[154] L. Wang, Y. Li, and K. Sun, "Amnesia: A Bilateral Generative Password Manager," in *36th IEEE International Conference on Distributed Computing Systems (ICDCS 2016)*, Nara, Japan: IEEE Computer Society, Jun. 2016, pp. 313–322.

[155] A. R. K. Moriarty B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.1," IETF, RFC 8018, Jan. 2017, p. 40.

[156] S. J. C. Pervical, "The scrypt Password-Based Key Derivation Function," IETF, RFC 7914, Aug. 2016, p. 16.

[157] H. Bojinov, E. Bursztein, X. Boyen, and D. Boneh, "Kamouflage: Loss-Resistant Password Management," in *15th European Symposium on Research in Computer Security (ESORICS 2010)*, Athens, Greece, Sep. 2010, pp. 286–302.

[158] E. Stobert and R. Biddle, "A Password Manager That Doesn't Remember Passwords," in *Proceedings of the 2014 New Security Paradigms Workshop (NSPW 2014)*, Victoria, British Columbia, Canada: Association for Computing Machinery, Sep. 2014, pp. 39–52, ISBN: 9781450330626.

[159] D. Reichl, *KeePass Password Safe*, https://keepass.info/, Sep. 2020.

[160] D. McCarney, D. Barrera, J. Clark, S. Chiasson, and P. C. van Oorschot, "Tapas: Design, Implementation, and Usability Evaluation of a Password Manager," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC 2012)*, Orlando, Florida, USA: Association for Computing Machinery, Dec. 2012, pp. 89–98, ISBN: 9781450313124.

[161] H. Li and D. Evans, *Horcrux: A Password Manager for Paranoids*, Oct. 2017. arXiv: 1706.05085 [cs.CR].

[162] F. Zinggeler, "NoKey - A Distributed Password Manager," M.S. thesis, ETH Zürich, Zürich, Switzerland, 2018.

[163] Y.-T. Liu, D. Du, Y.-B. Xia, H.-B. Chen, B.-Y. Zang, and Z. Liang, "SplitPass: A Mutually Distrusting Two-Party Password Manager," *Journal of Computer Science and Technology*, vol. 33, pp. 98–115, Jan. 2018.

[164] M. Eggimann and C. Gloor, "Convenient Password Manager," ETH Zürich, Zürich, Switzerland, Tech. Rep., 2016.

[165] M. Shirvanian, S. Jareckiy, H. Krawczykz, and N. Saxena, "SPHINX: A Password Store that Perfectly Hides Passwords from Itself," in *IEEE 37th International Conference on Distributed Computing Systems (ICDCS 2017)*, Atlanta, GA, USA, Jun. 2017, pp. 1094–1104.

[166] I. Krstić, "Behind the scenes with iOS security," in *Black Hat*, 2016.

[167] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: Bringing Key Transparency to End Users," in *24th USENIX Security Symposium (USENIX Security 2015)*, Washington, D.C.: USENIX Association, Aug. 2015, pp. 383–398, ISBN: 978-1-939133-11-3.

[168] Google, Inc., *Google Cloud Key Vault Service | Android Developers*, https://developer.android.com/about/versions/pie/security/ckv-whitepaper, Oct. 2020.

[169] ——, *Privacy Policy – Privacy & Terms – Google*, https://policies.google.com/privacy, Jan. 2020.

[170] T. Haselton, *How to find out what Google knows about you and limit the data it collects*, https://www.cnbc.com/2017/11/20/what-does-google-know-about-me.html, Dec. 2017.

[171] AdBlock, *Surf the web without annoying pop ups and ads!* https://getadblock.com/, Aug. 2020.

[172] eyeo GmbH, *Adblock Plus | The world's # 1 free ad blocker*, https://adblockplus.org/, Aug. 2020.

[173] gorhill, *gorhill/uBlock: uBlock Origin - An efficient blocker for Chromium and Firefox. Fast and lean.* https://github.com/gorhill/uBlock, Aug. 2020.

[174] L. H. Newman, *Google Turns to Retro Cryptography to Keep Data Sets Private | WIRED*, https://www.wired.com/story/google-private-join-compute-database-encryption/, Jun. 2019.