Victory by KO: Attacking OpenPGP Using Key Overwriting*

Lara Bruseghini
ETH Zurich and Proton AG
larabr@protonmail.com

Kenneth G. Paterson
Applied Cryptography Group, ETH Zurich
kenny.paterson@inf.ethz.ch

Daniel Huigens
Proton AG
d.huigens@protonmail.com

ABSTRACT

We present a set of attacks on the OpenPGP specification and implementations of it which result in full recovery of users' private keys. The attacks exploit the lack of cryptographic binding between the different fields inside an encrypted private key packet, which include the key algorithm identifier, the cleartext public parameters, and the encrypted private parameters. This allows an attacker who can overwrite certain fields in OpenPGP key packets to perform cross-algorithm attacks, causing a user's software to, for example, misinterpret an ECC private key as being a DSA key. It also allows an attacker to replace the legitimate public parameters with adversarially chosen ones, e.g. allowing them to select the DSA group. We refer to this class of attacks as Key Overwriting (KO) attacks. We provide a detailed analysis of the vulnerability of different OpenPGP libraries to KO attacks, showing in particular that in some cases additional key validation steps performed by libraries that should prevent the attacks in fact allow variant attacks. We also assess the applicability of KO attacks in the context of specific OpenPGP-based applications that reflect different threat models. Finally, we explain how KO attacks can be completely prevented (and the need for key validation obsoleted) at the OpenPGP specification level by expanding the existing proposal of using AEAD schemes for key packet protection to have all the security-relevant public fields included as Associated Data.

1 INTRODUCTION

OpenPGP [7] is a specification that was originally developed in the 1990s to secure general electronic communication and data. Nowadays, the protocol is mostly known for email encryption, but it is also widely used to secure storage and for data authentication through signatures. Starting with [35] in 1999 and continuing up to the present day [2, 11, 21, 34], OpenPGP has been heavily criticized on security and usability grounds. The security concerns stem from the complexity of the employed data format and from its reliance on cryptographic algorithms and constructions which are considered outdated and not provably secure. In addition to these purely technical concerns, securing emails using OpenPGP can entail significant usability issues for the average user [31, 32, 35].

However, various email applications now offer a more seamless user experience by making use of OpenPGP "under the hood", and there is an active ecosystem of developers, libraries and applications using the specification today. Indeed, OpenPGP remains one of the most widely used specifications for email encryption, and there are ongoing efforts to specify a new version of it [19, 20], aiming to update its underlying cryptographic primitives. This continued relevance justifies the ongoing analysis of OpenPGP.

Analysis of email systems is traditionally conducted in a weak threat model, where interactions with mail servers are limited to downloading and sending messages, and remote parties do not get to communicate directly with cryptographic software, but where that software is only used locally to decrypt/encrypt or sign/verify some emails. However, the use cases for OpenPGP have evolved, and application scenarios have changed over the past 20 years. In particular, we now see widespread use of cloud-based storage, in-browser and server-provided encryption services, and automated cryptographic processing by those services. Hence, modelling assumptions that were reasonable in the past need to be challenged and security requirements reassessed. Moreover, the OpenPGP standard is very flexible and does not target the specific use case of securing email. So, the OpenPGP specifications and its implementations cannot settle for weak adversarial models relevant for traditional email deployments, but should instead aim to defend against standard classes of attack.

In this paper, we focus on the consequences of storing encrypted private OpenPGP keys in insecure storage, and show how an attacker with write access to the corresponding encrypted key packet can tamper with it and extract the private key once the modified key packet is used. We also outline a conceptually simpler attack that allows accessing sent messages. The insecure storage threat model is relevant to some modern cloud-based applications using OpenPGP. For example, ProtonMail[1] allows users to access their data from multiple devices by storing each user's private key in encrypted form on the ProtonMail servers, with the encryption passphrase being unknown to the servers.

The aforementioned *Key Overwriting (KO)* attack vector is not new: as early as 2001, Klíma and Rosa [17] showed how to target DSA and RSA keys in this way. Their work led to key validation being introduced in some implementations of OpenPGP, as well as specification changes to prevent the attack for RSA keys. We revisit the core idea of [17] in the context of modern applications of OpenPGP where, for example, a server may have the capabilities needed to carry out a KO attack and should be considered untrusted.[2] We perform a systematic exploration of KO attacks and show how they can be used to extract private keys for *any* supported algorithm (i.e. EdDSA, ECDSA, ECDH, RSA, DSA and ElGamal). The widespread applicability arises from a cross-algorithm attack: the OpenPGP specification does not include cryptographic mechanisms that allow systematic detection of certain types of private key packet corruption, and does not require that the private key be cryptographically bound to its type. This means that the security of any key type is reduced to the security of the weakest key type that has interchangeable private parameters. Specifically, since we provide efficient attacks for both DSA and ElGamal keys, we obtain efficient attacks for the other discrete logarithm key types,

*To appear in ACM CCS 2022.

[1] https://protonmail.com/

[2] Indeed, *not* requiring trust in the server for security is part of the advertised appeal of some cloud-based applications, see for example https://protonmail.com/blog/protonmail-threat-model.

namely EdDSA, ECDSA and ECDH keys. We also provide RSA-specific attacks, both for the signing and encryption settings; these attacks require the use of non-CRT-based private key operations.[3] We stress that KO attacks assume that users (and the applications they use) are not careful in constantly checking the fingerprints of their own keys (and instead trust the keys if they can be decrypted with the user's passphrase, for example) – such vigilance would prevent the attacks.[4]

As mentioned above, some OpenPGP libraries carry out key validation steps before using keys. The steps are not specified in the OpenPGP specification, and so are implementation- as well as algorithm-specific. In principle, proper key validation would prevent our attacks, and so we evaluate how key validation is carried out in different OpenPGP libraries and to what extent it hinders our attacks. We find significant diversity in how key validation is performed – some libraries do none, while others are quite thorough. Nevertheless, we also show that, in several cases, improper key validation in combination with key overwriting opens up a new class of attack which we call a *Key Overwriting Attack Exploiting Key Validation (KOKV attack)*. Such an attack can also result in private key extraction.

To make our attacks concrete, we show how the attacks described above in the context of OpenPGP libraries can be realized for two specific OpenPGP-based applications making use of those libraries: FlowCrypt and ProtonMail. Finally, we also consider immediately deployable and longer-term countermeasures to KO (and KOKV) attacks: the former simply demands that implementations do careful key validation, while the latter relies on AEAD encryption to protect private keys. Following our disclosure, the long-term solution has been incorporated in the draft revision of the OpenPGP specification [20].

Vulnerability Disclosures. We contacted the OpenPGP Working Group as well as the maintainers of all the libraries reviewed in this paper regarding our attacks between November 2020 and January 2021. The OpenPGP.js and gopenpgp libraries have since been patched to perform algorithm-specific parameter checks as part of key decryption.[5] RNP also shipped full attack countermeasures in their recent v0.16 release.[6] GnuPG and Sequoia do not plan to implement any changes: the GnuPG developers believe that users should not rely on the security of the key encryption mechanism when storing or transferring the keys (and instead e.g. send keys encrypted in a proper OpenPGP message), while the Sequoia developers consider key storage attacks as out of scope for their threat model. At the application level, both FlowCrypt and ProtonMail have been updated to rely on secured versions of their underlying OpenPGP libraries (OpenPGP.js and gopenpgp). Hence, at the time of writing, the application-level attacks described in Section 4 are no longer possible.

Related Work. As noted above, the idea of long-term private key compromise through overwriting of the encrypted key was originally presented in [17], with attacks targeting RSA and DSA keys showing how to corrupt the victim's encrypted private key to then recover the secret exponent once the corrupted key was used for signing. The RSA attack exploited a weakness in the OpenPGP key encryption mechanism, and the OpenPGP specification was changed in RFC 4880 [7] to fix the issue by introducing a new integrity mechanism (see comments about *string-to-key usage 254* in [7, Section 5.5.3]). The DSA attack exploited a different problem: the attacker does not tamper with the encrypted data, but rather with the cleartext values included alongside it inside the key packet; [17] showed how a signature generated with such a corrupted key could leak the DSA secret used.

A significant number of other papers have considered different security aspects of OpenPGP. A particular focus has been chosen-ciphertext attacks (CCA) against OpenPGP's message encryption construction. Nguyen [26] pointed out that RSA and ElGamal encryption do not achieve CCA security as they both use PKCS#1 v1.5 padding [15], which is vulnerable to Bleichenbacher's attack [4]. Other works showed how messages could be recovered by exploiting some oracles exposed when processing unauthenticated data in non-constant-time [23, 25] or by tricking the victim into sharing the seemingly-random decrypted data [13, 16]. The EFAIL attack [28] showed how plaintext could be exfiltrated by corrupting encrypted messages and exploiting the mishandling of integrity checks by applications. None of this work targets private key recovery. The recent paper [6] focuses on the insecurity of ElGamal encryption in OpenPGP implementations. Interestingly, it considers insecure interactions between different implementations of the ElGamal algorithm, whereas we consider cross-algorithm rather than cross-implementation attacks.

Paper Structure. We provide background information on OpenPGP in Section 2. In Section 3 we describe our KO attacks, by presenting the theoretical details and then assessing the status of some popular OpenPGP libraries with respect to our attacks. In Section 4 we review the impact and practicality of the attacks in the context of OpenPGP-based applications, reflecting different threat models. Section 5 discusses short-term and long-term countermeasures against our attacks while Section 6 contains our closing remarks.

2 BACKGROUND ON OPENPGP

The OpenPGP specification aims to provide confidentiality and authenticity for electronic communications: its function is to "provide data integrity services for messages and data files" [7]. It is widely used as an email encryption standard, but it is also a popular solution to secure data at rest.

The OpenPGP specification was published in 1998 as RFC 2440 [8], later superseded by RFC 4880 [7]. This last RFC is complemented by RFC 6637 [14] which adds support for Elliptic Curve cryptography. Since 2015, a new version of the standard has been under development, with most of the changes proposed as part of the draft RFC 4880bis [19]. The OpenPGP Working Group was recently reactivated in the IETF and is chartered to complete the work on

[3] RSA private keys should not be convertible to the other types, since RSA keys contain multiple private parameters but all the other key types contain only one parameter. Hence, we attack RSA private keys separately.

[4] We also assume that the user has picked a strong enough key passphrase, otherwise the attacker could simply run a fast dictionary attack to decrypt the key [24].

[5] See https://github.com/openpgpjs/openpgpjs/pull/1116 and https://github.com/ProtonMail/go-crypto/pull/59.

[6] See https://github.com/rnpgp/rnp/commit/f63f9849cc2cedb06cb35ee8f2fb0d919bc2e6da.

a new version of the standard. At the time of writing in October 2021, the work to revise the specification is ongoing.[7]

Draft RFC 4880bis. Besides updating the cryptographic algorithms used, the RFC 4880bis draft introduces new features, such as Authenticated Encryption with Associated Data, support for additional elliptic curves, and a new (version 5) key format. Even though these changes have not been standardized yet, many implementations have started to support them. In this paper we will focus on the draft specification RFC 4880bis-10 [19][8]. Note that all the issues presented in this paper also affect the standard specification (except for the attacks targeting EdDSA keys, which were introduced in RFC 4880bis).

Examples of OpenPGP-based Applications. OpenPGP enables email encryption, and it is supported by many email clients either natively or via plugins. For instance, Mozilla Thunderbird has recently added (experimental) native support for OpenPGP encryption; FlowCrypt and Mailvelope are browser extensions that integrate OpenPGP encryption in a number of third-party email services, including Gmail. In another deployment model, ProtonMail is an email provider that offers out of the box end-to-end encryption to its users using the OpenPGP specification. This makes the service interoperable, allowing secure communication with third-party email addresses. Additionally, OpenPGP is employed in backup solutions to secure files at rest. It is also used in settings that solely require authenticity services. For example, the Debian package manager aptitude (apt) uses OpenPGP signatures to verify the legitimacy of packages being distributed. Similarly, git commits can be signed using OpenPGP keys to identify the author of the code and prevent forgeries.

Supported Cryptographic Functions. OpenPGP specifies methods to encrypt and sign messages, relying on long-term keys for authentication. It also supports certification functions in order to help confirm the identity of the user bound to a given key based on the "web of trust".

For encryption, a message is always encrypted symmetrically with a session key, which in turn is encrypted to a public key or using a key derived from a passphrase. Historically, symmetric encryption relied on the CFB mode of operation. To protect the integrity of the plaintext, RFC 4880 introduced an optional mechanism called Message Detection Code (MDC). The MDC is essentially a SHA-1 hash of the plaintext that is appended to the plaintext packet. Support for authenticated encryption (AEAD) primitives was introduced in [19]. This draft also makes the MDC mandatory when not using AEAD.

User IDs and the Web of Trust. OpenPGP users are identified by "user IDs", which typically contain an email address. Zero or more User IDs can be attached to a given key, but there is no built-in system for identity verification; it is ultimately up to the individual users to decide whether a given key belongs to the declared entity. To help determine the authenticity of the stated user IDs, other users are expected to certify entities who they know own a given key. Anyone can endorse such an association by issuing a third-party certification signature over the key and the relevant user ID. In principle, this form of trust is transitive, and the resulting network is referred to as the web of trust.

Packet and Message Structure. OpenPGP uses a packet based format: an OpenPGP message is made up of concatenated and nested packets of the form `tag||length||body`. The `tag` field identifies the packet type, while the `length` field specifies the size of the body in octets. The content of the body is specific to each packet type. There are 18 different packet types which can be combined in several ways, even though not all combinations are meaningful. Long-term keys are also made up of multiple packets.

Signatures. OpenPGP signatures are encoded using the Signature packet type, which has different uses depending on the kind of data being signed [19, Section 5.2]. Aside from guaranteeing the authenticity of OpenPGP messages, this packet type is used for certification of keys and users, but also to augment keys with algorithm preferences and other features.

A Signature packet is made up of subpackets which add information to the signature. Example of subpackets are: the creation date of the signature, or the fingerprint of the issuer. When generating a signature, a digest is computed over the data to sign as well as part of the Signature packet body, including the specific subset of the subpackets that are marked as "hashed subpackets". In fact, not all subpackets have to be hashed (and thus authenticated). The reason for this is unclear and may be historical (e.g. at one time, it might have been too expensive to sign the entire packet). The OpenPGP specification indicates a few subpackets that must be hashed, most notably the creation time, as well as the signature character set (if present).

2.1 Long-Term Keys

Long-term keys are used to provide the confidentiality and authenticity described in the previous sections. To communicate securely with someone, their public key needs to be known. Similarly, to verify signatures, one needs to have the corresponding public key and trust that it belongs to the stated entity. There are two supported versions of key packets: version 4 and 5. These differ in terms of the stored metadata. Unless otherwise stated, our observations apply to both.

Public Key Packet. A Public Key packet includes the public key parameters[9] necessary for encryption and signature verification. A public key is identified by its fingerprint, which is computed as the hash of the entire key packet (SHA-256 is used for v5 keys and SHA-1 for v4 keys).

Secret Key Packet. A Secret Key packet is an extension of the Public Key packet which also includes the private parameters and a mechanism to protect them [19, Section 5.5.3]. The private parameters can be encrypted using a key derived from a passphrase via a string-to-key (S2K) process (whose details are immaterial to

[7] The latest progress can be determined from https://datatracker.ietf.org/wg/openpgp/.

[8] RFC 4880bis was the latest draft available when the majority of this work was being carried out. The OpenPGP Working Group has since started a new document, named crypto-refresh [20], which is set to incorporate most of the changes proposed in RFC 4880bis. The content of this paper does not apply to the latest crypto-refresh draft (version 04), which was updated to address our attacks following our disclosure.

[9] With "key parameters" or "key material" we refer to the algorithm-specific values that comprise a key (either public or private). From now on, we will use the terms "public key" and "private key" to indicate the corresponding OpenPGP packet, which includes the key parameters, as well as other information.

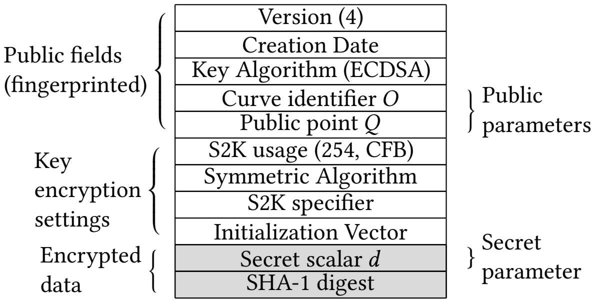

Figure 1: Secret Key packet body of an ECDSA key (version 4) protected using CFB mode. The SHA-1 digest is computed over the secret parameters. Notice that the public parameters are not cryptographically bound to the secret one.

our attacks). The encryption uses either CFB mode or AEAD. If the former is used, the private parameters are encrypted together with a 20-byte SHA-1 hash of their values, intended to protect their integrity. Regardless of the protection mechanism used to encrypt the secret key, only the private parameters are cryptographically protected. In fact, the public parameters are always stored in cleartext and are not cryptographically bound to the (encrypted) private ones. This is illustrated in Fig. 1, which shows the structure of a version 4 ECDSA Secret Key packet body.

Key and Subkey Packets. An OpenPGP key consists of a Secret or Public Key packet ("primary key") and any number of Subkey packets. Secret and Public Subkey packets have the same body format as Secret and Public Key packets respectively, and only differ in their packet tag. Subkeys are typically used for encryption operations, whereas primary keys are used for signing and certification functions (of subkeys and User IDs). Subkeys are stored alongside their primary key, but each packet is protected separately.

Certification Signatures. Public keys and subkeys must be signed by the corresponding private primary key to be used in practice, otherwise they are not considered trusted by implementations. A primary key certifies that a subkey is associated with it by issuing a Subkey Binding Signature computed over the subkey's public key. In turn, signing subkeys must sign back their primary public key via Primary Key Binding Signatures. Finally, many implementations require that keys be associated with at least one User ID through a self-certification signature (of type 0x10-0x13), issued by the primary key. For more details, see [19, Section 5.2.1].

2.2 OpenPGP Implementations

Several implementations of the specification have been developed. When discussing the practical impact of attacks in this paper, we will focus on the following open-source libraries. This is a representative sample of implementations from different languages and used in popular applications, including those for which we discuss fully developed attacks in Section 4.

GnuPG (GPG): the OpenPGP distribution of the GNU foundation. It is the oldest and most popular open source implementation, written in C.

RNP (prior to v0.16[10]): a C++ library used by Thunderbird for their native OpenPGP integration.

OpenPGP.js (prior to v4.10.5[10]): a JavaScript OpenPGP library used by ProtonMail for their web app, as well as by FlowCrypt and Mailvelope. It offers two different distributions: a browser version and a server-side Node.js one. The two versions use different cryptographic libraries, hence their behaviour in some cases can vary. In this paper, we only look at the browser version.

gopenpgp (prior to v2.1[10]): a golang OpenPGP library used by ProtonMail for their desktop and mobile apps.

Sequoia PGP: a newer OpenPGP implementation, written in Rust. Sequoia supports different crypto backends, but for this paper we only consider Nettle as the underlying crypto library, as it is the default one.

3 PRIVATE KEY EXTRACTION VIA KEY OVERWRITING

As introduced in Section 2.1, the public key material stored as part of the secret key packet is always in cleartext (see Fig. 1). Moreover, said material is not cryptographically bound to the private parameters stored in the ciphertext part of the secret key packet. We show how to exploit these properties to perform KO attacks that recover the private parameters when the corrupted key is used for either signing or decrypting. In both cases, we exploit the interactions between the victim's private parameters and the overwritten public ones to learn information about the former. When signing, using corrupted parameters results in so-called "faulty signatures", which can allow private key reconstruction; when decrypting arbitrary ciphertexts using malicious parameters, the outcome (success/failure) of the operation can leak information about the private parameters involved.

We start by introducing our threat model, and tackle some technicalities that an adversary needs to deal with in practice. Then we describe KO attacks against the OpenPGP specification for an adversary that has write access to encrypted key packets and that is able to corrupt their contents. For each attack, we study how different libraries are affected. Most of them perform some form of key validation, which is not covered by the OpenPGP specification but could prevent our attacks. However, we show that the implemented steps are often ineffective and, in some cases, have observable side effects that make it possible to exploit the validation process itself to carry out KOKV attacks. We describe those KOKV attacks in Appendix C.

3.1 Threat Model

We consider an adversary who has write access to the encrypted private key of the victim. Further, we assume that the victim (resp. their application) will use the private key as long as it decrypts successfully with the expected passphrase. In other words, we assume that the victim does not inspect their own key fingerprint before using the key.[11]

[10] For RNP, OpenPGP.js and gopenpgp, the results in this paper only apply to the specified older versions; following our disclosures, the libraries have been patched against the vulnerabilities presented in this paper.

[11] Checking the key fingerprint would always reveal whether the key was corrupted, but users might not expect to have to check its value for their own keys (or might not

This threat model is especially relevant when it comes to services that manage the user's encrypted private keys and store them in insecure storage, in the sense that the key files might be accessible to untrusted parties. This is the case for applications like ProtonMail and FlowCrypt, which we review in Section 4. However, in principle keys can also be targeted when they are, for example, stored on a USB drive, stored in the cloud, or are in transit.

In addition to corrupting the encrypted keys, depending on the attack in question, the attacker also needs to get hold of signatures generated by the victim, or learn the success/failure of some message decryption. Section 4 covers application-level considerations and explains how the adversary can access this type of information to carry out end-to-end attacks.

3.2 Practical Considerations

Key Validation. Many OpenPGP libraries add key validation checks that aim to verify the mathematical relationship between public and private parameters before using them in any cryptographic operations. Such validation is not covered by the OpenPGP specification. Since our attacks modify public parameters, proper key validation could prevent them from being carried out in practice. For this reason, after presenting each attack, we review which implementations are actually vulnerable. We now give a brief overview of what key validation is done by each library. For full details, see Appendix D. Sequoia does not implement any checks, while all the other libraries we study do: GPG and RNP check some mathematical relationships between the parameters, whereas OpenPGP.js and gopenpgp try to sign and verify a message using the primary key. However, key validation is not necessarily carried out by these libraries. In fact, GPG and RNP validate the keys when they are imported into the corresponding key stores. In GPG the user must import the keys before usage, hence key validation is always run. However, RNP does not enforce the use of the key store, hence some attacks are viable depending on how an application loads the user keys. Finally, OpenPGP.js and gopenpgp do not automatically check private keys, but simply provide a key validation function that can be called by applications, if desired.

Bypassing Key Certification Signatures. In Section 2.1, we explained how OpenPGP keys need to include a number of certification signatures verifiable with the primary public key in order to be considered valid by most implementations. Hence, for KO attacks to work in practice, in addition to corrupting the public parameters, the adversary needs to provide forged certification signatures that can be verified using them. We explain how to do so for the non-obvious cases in Appendix B.1.

3.3 KO Attacks Exploiting Faulty Signatures

Our first KO attack consists of tricking the victim into signing a message using their corrupted key. From the resulting faulty signatures, the adversary is able to learn at least partial information about the private parameters. We detail how keys of any algorithm can be targeted, by replacing legitimate public key parameters with malicious ones – notation wise, we always denote the latter with an apostrophe, e.g. p' is the replacement value of some parameter p. We start by introducing two attacks against DSA keys.

3.3.1 *Faulty Signature Attacks against DSA.*

The DSA signing algorithm is shown in Algorithm 1 and uses the private exponent x together with the public parameters g, q, p where p is a prime, q is a large prime factor of $p-1$ and g is an element of $\mathbb{Z}_p^*$ with order q. The attack from [17] can be used to extract the private parameter (key) through a single faulty signature. However, that attack no longer works in practice since it requires replacing the group prime p with a corrupted value p' such that $p' < q$, where q is typically 160 or 256 bits long, while DSA keys with p of 1024 bits or less are no longer supported by several implementations, since they are deemed insecure [27]. Thus, we describe two alternative attacks that can still be carried out against most libraries.

Variant 1: Klíma-Rosa attack revisited. The private key can be computed from one faulty signature as follows: let p' be a prime of the desired key size such that $p'-1$ is smooth, then set q' to be the next prime above p' and g' to be a generator of $\mathbb{Z}_{p'}^*$. The adversary overwrites g, p, q in the victim's key packet with g', p', q', and then needs to obtain one signature generated using the corrupted key. From such a DSA faulty signature (r, s) over message m, the secret x can be reconstructed as follows: since $q' > p'$, we have that $r' = (g'^k \bmod p') \bmod q' = g'^k \bmod p'$, and we can recover k by computing the discrete log of r' to base $g' \bmod p'$. Finding the discrete logarithm is feasible as p' is smooth by construction, and allows us to recover $k \bmod p'-1$. Given k, we can compute $x = (s \cdot k - Hash(m)) \cdot r^{-1} \bmod q'$. We now give more details about the correctness of the attack. Thanks to our choice of q' we are almost guaranteed to recover k in full: while k is originally sampled in $Z_{q'}$, which means that k might be larger than $p'-1$, in our case $k \bmod p'-1 = k \bmod q'$ unless $k \in [p', q'-1]$. The probability of sampling such k is on the order of $(\log p')^2/p'$.[12] So this probability is small enough that it can be ignored in our attack.

Variant 2: small subgroup attack. The attack described above allows extraction of the private key through one single faulty signature. Still, due to the large size of q', the corrupted key may not be supported by some implementations, as we will see later. Hence, we describe a second attack on DSA keys using a small subgroup attack. This technique requires tens of signatures to reconstruct the full secret, but it gives the adversary more flexibility. The adversary overwrites the primes p, q and the subgroup generator g with some prime p' and small prime q' such that g' has order q' in $\mathbb{Z}_{p'}^*$. From the resulting faulty signature (r, s), the adversary can recover $x \bmod q'$ by finding the value x^* such that using $y' = g'^{x^*} \bmod p'$ in the verification of (r, s) is successful. The search for x^* can be done offline. The original $x \bmod q$ can be reconstructed fully by collecting enough signatures computed using different p'_i, q'_i and combining the different shares $x^* \bmod q'_i$ using the Chinese Remainder Theorem (CRT). Finding the value of $x \bmod q'_i$ fails in two cases: (1) if $g'^k \bmod p'$ is a multiple of q', since then $r = 0 \bmod q$,

check it thoroughly enough). Further, to lower the chance of detection if the victim only looks at a few hex fingerprint digits, it should be feasible to forge a public key whose fingerprint matches the original one in the first and/or last 3-4 bytes: one could keep the public parameters fixed and change the creation date (in seconds) to get different fingerprints. For more details on the fingerprint, see Section 2.1.

[12] q' is set to be the next prime above p' and the gaps between primes are, according to Cramer's conjecture $O((\log p')^2)$.

Algorithm 1: DSA signing and verification

```
Data: Message $m$, public key $(g, p, q, y)$, private key $x$,
      signature $(r, s)$
Function DSA_Sign($m$, $(g, p, q), x$)
    $k \xleftarrow{\$} [1, q-1]$
    $r = (g^k \bmod p) \bmod q$
    $s = k^{-1}(\mathrm{Hash}(m) + xr) \bmod q$
    return $(r, s)$
Function DSA_Verify($m$, $(g, p, q, y), (r, s)$)
    $w = s^{-1} \bmod q$
    $h = \mathrm{Hash}(m) \bmod q$
    $v = (g^{hw} y^{rw} \bmod p) \bmod q$
    return $v \overset{?}{=} r$
```

and thus any y' checks against the signature; or (2) if q' is a factor of s, meaning that the inverse $w = s^{-1} \bmod q'_i$ does not exist, and the signature is not verifiable. Using q'_i which are not too small lowers the probability of failure. Larger primes are also preferable as they lower the number of key overwrites and faulty signatures needed in order to fully reconstruct x. For example, a 256-bit x can be computed from about 16 key overwrite and faulty signatures provided the adversary picks each q'_i to have 16 bits. At this size of q'_i, finding the individual values x^* is still efficient. This is a typical size for x for 2048-bit DSA public parameter p.

Vulnerable Implementations. Note that in practice, the second attack variant might not work exactly as detailed above, since most libraries expect q to be at least 160 bits long. To get around this limitation, it suffices to multiply q' by some co-factor h of an appropriate size, and overwrite q with $q'' = q'h$ but still use g of order q' in the attack. When it comes to the security of individual libraries, Sequoia is vulnerable to both attacks, since it does not carry out any key validation. OpenPGP.js and gopenpgp are also vulnerable, since their key validation can be bypassed by setting a DSA primary key with $g = y = 1$ and adding the target key as subkey.[13] RNP and GPG are safe against the first attack: GPG is not vulnerable thanks to its key validation, whereas RNP does not support using q larger than 256 bits. As for the second attack variant, the key validation in RNP is sufficient to prevent the attack, whereas the key validation in GPG is incomplete and can only slow down the adversary, requiring thousands of key overwrites instead of a few tens of key overwrites. In fact, while both libraries verify the correspondence between the public and private parameters, GPG does not confirm whether the group order is prime and large enough. Hence, the adversary can successfully import a corrupted key in GPG by guessing y' such that $y' = g'^x \bmod p'$. Additionally, the partial key validation done in GPG (as well as in OpenPGP.js and gopenpgp) opens up the possibility of exploiting the key validation process itself to recover the private key. This attack vector is discussed in Appendix C.

Algorithm 2: EdDSA signing algorithm

```
Data: Message $m$, curve base point $G$ of order $q$,
      secret seed $k$, signature $(R, S)$
Function EdDSA_Sign($m$, $(G, q), k$)
    $(d, p) = \mathrm{Hash}(k)$        // scalar $d$ and prefix $p$
    $Q = dG$                          // derive public point $Q$
    $r = \mathrm{Hash}(p||m)$
    $R = rG$
    $h = \mathrm{Hash}(R||Q||m)$
    $S = r + d \cdot h \bmod q$
    return $(R, S)$
```

3.3.2 Cross-algorithm Attack against ECC keys. The above attacks on DSA are quite impactful as they can be used to indirectly compromise ElGamal and ECC keys (i.e. EdDSA, ECDSA and ECDH keys). These can be "converted into" DSA ones by keeping the encrypted key material and replacing the public key parameters and key algorithm to reflect DSA keys. This trick is possible because all the keys in question have a single private parameter, hence the format of the encrypted portion of the key is the same. It follows that ECC keys are vulnerable to the same set of attacks that target DSA keys, including the ones described above.

3.3.3 Faulty Signature Attack against EdDSA. It is known that in EdDSA, the secret scalar d can be recovered from two signatures generated over the same message but using different public keys.[14] For reference, the EdDSA signing algorithm is given in Algorithm 2. Assuming that the public point Q is taken directly from the supplied public key and not re-derived from d, then signing the same message m with both Q and Q' gives two different signatures (S, R) and (S', R') with $S = r + dh \bmod q$ and $S' = r + dh' \bmod q$, from which we can find d as: $d = \frac{S-S'}{h-h'} \bmod q$, where h, h' are known. Note that in OpenPGP, the message m includes the data being signed, as well as additional metadata, such as (a) signature creation time and (b) (optional) issuer public key fingerprint. If the fingerprint is signed then the attack cannot be executed, because m itself will depend on the public key, so we cannot arrange to sign the same m using different Q values. If only the creation time is signed, then the attack can still succeed if the adversary can get two signatures with the same timestamp. This may be feasible, since the creation time is taken in seconds and the adversary might be able to execute commands concurrently or control the time seen by the application.

Vulnerable Implementations. While GPG and Sequoia do not validate EdDSA keys, at the time of writing they are both protected, since by default they do include the Issuer Fingerprint in signatures. Signing the Issuer Fingerprint is not mandated by the specifications and if the libraries were to omit (or not sign) the field/subpacket for any reason, the attack would be possible. OpenPGP.js and gopenpgp do not sign the Issuer Fingerprint, and are thus vulnerable if key validation is not performed (this is up to the application). Finally, RNP is always safe since it re-derives the point Q from d before signing.

[13] If the target key is a primary key, its Secret Key packet can be converted into a Secret Subkey packet just by changing its packet tag. This trick can also be used to obtain a DSA primary key from a DSA subkey, in case a DSA primary key is needed. Note that this works for standard CFB-encrypted keys, but not for the proposed AEAD-encrypted keys when the target is a primary key, because AEAD-encrypted keys do authenticate whether the encrypted data is from a primary key or a subkey.

[14] See discussion in the libsodium repository at https://github.com/jedisct1/libsodium/issues/170. There was no actual vulnerability in the library since the public point was re-derived from the secret scalar.

	DSA v1 (+ ECC)	DSA v2 (+ ECC)	EdDSA	RSA
Key Overwrites	1	15-bit q_i 256-bit x: 18 512-bit x: 35	1	1
Signatures	1	256-bit x: 18 512-bit x: 35	2	1

Table 1: Cost of the different faulty signature attacks in terms of key overwrites and number of signatures needed to compute the secret.

	DSA v1 (+ ECC)	DSA v2 (+ ECC)	EdDSA	RSA
GPG	no	depends	no (fingerprint)	no (CRT-RSA)
Sequoia	yes	yes	no (fingerprint)	no (CRT-RSA)
RNP	no	depends	no (fingerprint)	no (CRT-RSA)
OpenPGP.js	yes	yes	depends	depends
gopenpgp	yes	yes	depends	no (CRT-RSA)

Table 2: Vulnerability status of the different implementations with respect to faulty signature key extraction attacks (yes: vulnerable, depends: vulnerable unless the key is explicitly validated or based on other factors described in this section). If a library is not vulnerable, but not primarily due to key validation, then the reason is specified.

3.3.4 Faulty Signature Attack against Non-CRT RSA. Consider RSA signatures in which no CRT tricks are used to speed up signing. The adversary can replace n with a number n' for which $\varphi(n')$ is smooth. From the resulting signature $s' = m^d \bmod n'$, the adversary can recover d by computing $\log_m(s')$ via the Pohlig-Hellman algorithm. To recover d in full, the order of m in $\mathbb{Z}^*_{n'}$ must be large enough, namely $ord_{n'}(m) \geq d$. If m is known in advance, the adversary can pick n' such that m has full order. Even if not, the condition $ord_{n'}(m) \geq d$ will hold with high probability if we pick n' to be a prime such that the odd factors of $\varphi(n') = n' - 1$ are all mid-sized primes $\sim 2^{20}$. With this choice of n', solving the discrete logarithm problem is still easy.

Vulnerable Implementations. All implementations but OpenPGP.js use CRT-RSA for signing, hence the attack does not apply to them. Applications that rely on OpenPGP.js and do not explicitly validate keys are vulnerable. The key validation implemented by OpenPGP.js is sufficient to prevent this specific attack provided the attacker cannot predict at which time the key validation steps will be run. More details about this limitation, as well as how the validation process can be exploited to recover the private key in another way can be found in Appendix C.2.

3.3.5 Summary. We have shown how any key is potentially vulnerable to faulty signature attacks, either directly or indirectly (i.e. through cross-algorithm attacks). In particular, encryption-only keys such as ECDH and ElGamal keys can be compromised by converting them into DSA keys. The same is true for ECDSA keys, which are not directly impacted by fault attacks since the public point is not used during signing. We highlight that most of the attacks we have described require corrupting the victim's key only once, as shown in Table 1. For a summary of which implementations are vulnerable to each attack, see Table 2.

3.4 KO Attacks Exploiting Decryption

Another way to learn information about the private key – after corrupting the public parameters – is to observe whether decryption of chosen ciphertexts is successful. The idea is to exploit the decryption success or failure as an oracle to incrementally allow an adversary to recover the secret parameters. We stress that we do not require access to the decrypted data; the adversary only needs to know whether any errors occurred during the decryption operation. Depending on the application, accessing such errors may be easier than getting hold of faulty signatures. Further, as we will see, all the attacks in this section only require a single key overwrite.

We first formalize the oracle and then show how it can be used to exploit ElGamal and RSA decryption to recover the private parameters by means of small subgroup attacks. To minimize the number of key overwrites, we leverage the fact that OpenPGP supports adding an unlimited number of subkeys to a given primary key.

As with the faulty signature attacks, we review the vulnerability status of libraries immediately after describing each attack. Note that for such analysis we only consider whether there is sufficient key validation in place to prevent the issues, but we do not take into account whether accessing the decryption oracle is possible at an application level, since this depends on how the library is integrated. As we explain below, a library will typically expose the oracle through timing leakage. For application-level considerations, see Section 4.

Session-Key Decryption Oracle. In OpenPGP, long-term keys are not directly used to encrypt messages. In fact, messages are always encrypted symmetrically using a session key, while the session key is encrypted under a public-key and encoded in a Public-Key Encrypted Session Key (PKESK) packet. For the attacks in this section, we focus on the decryption of PKESK packets. Formally, we consider a decryption oracle that accepts the ciphertext c from a Public-Key Encrypted Session Key Packet and returns 1 if the decryption of c was successful, or 0 otherwise.

It is enough to consider as input to the oracle only a PKESK packet (namely without the subsequent symmetrically-encrypted message) for two reasons: firstly, both ElGamal and RSA encryption use PKCS#1 v1.5 padding (as per `EME-PKCS1-v1_5` in [15]), and secondly, the PKESK plaintext includes an algorithm byte, the session key, and a 16-bit checksum on that session key (see [7, Section 5.1]). The redundancy provided by these two encoding steps done in succession means that the probability of getting a false negative from the oracle, namely some c being marked as valid when it is not, is on the order of 2^{-48}. Thus, we can be sure that the oracle's results are correct without relying on a later symmetric decryption step failing due to a wrong session key being returned by the decryption of the PKESK packet.

Having formalized the oracle, let us make a practical remark about how it can be instantiated in practice, through timing leakage. Note that successful decryption of the PKESK packet is necessarily followed by symmetric decryption of the subsequent encrypted message. On the other hand, if PKESK decryption fails, in typical implementations, no symmetric decryption will take place. Hence, even if PKESK decryption is performed in constant time, it is likely that the oracle can be accessed through a timing side-channel exposed by the presence or absence of a symmetric decryption step.

Additionally, since the adversary controls the ciphertext, the timing leakage can be amplified by using large messages.

3.4.1 Decryption Oracle Attack against ElGamal. The ElGamal encryption and decryption processes are shown in Algorithm 3. The pseudo-code omits checking the 16-bit checksum on session keys for ease of presentation. The ElGamal public parameters include a prime p, a group generator g and $y = g^x \bmod p$. The only secret parameter is the exponent x, which is the target for our attack. We write $x = x_0 + x_1 \cdot 2 + x_2 \cdot 2^2 + \cdots x_{t-1}2^{t-1}$ where t is the known bit-length of x. We will recover the bits $x_0, x_1, \ldots, x_{t-1}$ in sequence.

The adversary begins by selecting a prime p' of the form $2^t h + 1$ and of the appropriate size. Such a prime is easy to construct by trial and error over values of h of the right bit-size. Having found p', the adversary constructs g_t of order $2^t \bmod p'$.[15] Now the adversary overwrites the victim's public parameters p, g, y with p', g', y'. Here g' and y' can be arbitrary, since they are not used on decryption. Hence, we simply select them so as to be able to forge key certification signatures as explained in Appendix B.1.

To recover x_0, the adversary sets $g_1 = g_t^{2^{t-1}}$, so g_1 has order 2. It selects m to be an arbitrary, validly encoded session key. It then sets $c_0 = g_1$, $c_1 = \mathsf{pad}(m)$, and submits $c = (c_0, c_1)$ to the decryption oracle. Now the decryption process run by the oracle sets $m' = c_1 \cdot c_0^{-x} \bmod p'$, yielding $m' = \mathsf{pad}(m) \cdot g_1^{-x} \bmod p'$. Since g_1 has order 2, we see that $m' = \mathsf{pad}(m)$ if and only if $x = 0 \bmod 2$, i.e. if and only if $x_0 = 0$. Thus, decryption of (c_0, c_1) succeeds if and only if $x_0 = 0$.[16]

To recover x_1, the adversary now sets $g_2 = g_t^{2^{t-2}}$ of order 4. It then sets $c_0 = g_2$, $c_1 = \mathsf{pad}(m) \cdot g_2^{x_0}$ with m as before, and submits (c_0, c_1) to the decryption oracle. Note the extra term $g_2^{x_0}$ in c_1 here. The oracle sets $m' = c_1 \cdot c_0^{-x} = \mathsf{pad}(m) \cdot g_2^{x_0} \cdot g_2^{-x_0-2x_1} = \mathsf{pad}(m) \cdot g_2^{-2x_1} = \mathsf{pad}(m) \cdot g_1^{-x_1} \bmod p'$. (Here we use the fact that g_2 has order 4 to ignore all but bits 0 and 1 of x). We see that $m' = \mathsf{pad}(m)$ if and only if $x_1 = 0$. Thus, decryption of (c_0, c_1) succeeds if and only if $x_1 = 0$.

The adversary proceeds in this way to recover x bit-by-bit. In the general case, to recover bit x_j, it sets $g_{j+1} = g_t^{2^{t-j-1}}$ of order 2^{j+1}, sets $c_0 = g_{j+1}$, $c_1 = \mathsf{pad}(m) \cdot g_{j+1}^{x_0+2x_1+\cdots x_{j-1}2^{j-1}}$, and sends $c = (c_0, c_1)$ to the decryption oracle, with successful decryption indicating $x_j = 0$.

The overall cost of the attack is one key overwrite and t oracle queries, where t is the bit-length of the private parameter x. In practice, in ElGamal, the bit-length of x is often much smaller than that of p, for efficiency reasons (e.g. GPG samples a 338-bit x for a 2048-bit p).

Notice that the above attack can be prevented by checking that $p-1$ is not divisible by a large power of 2 before using it (i.e. during a key validation step). However, there is a related small subgroup attack in which the prime 2 is replaced by a sequence of small primes q' dividing $p'-1$ and the CRT is used to reconstruct the private key. Such attack is more costly to mount in terms of decryption queries, but protecting against it is considerably more complicated. In particular, key validation would need to check that $p-1$ is not divisible by all small primes up to a certain size. See Appendix D for further discussion of the full ElGamal key validation and its cost.

Algorithm 3: ElGamal encryption and decryption with $\mathsf{pad}(\cdot)$ and $\mathsf{depad}(\cdot)$ denoting PKCS#1 v1.5 encoding and decoding.

```
Data: Message $m$, public key $(g, p, y)$, private key $x$,
      ciphertext $(c_0, c_1)$
Function ElGamal_Encrypt($m$, $(g, p, y)$)
    $k \xleftarrow{\$} [1, p-1)$
    $c_0 = g^k \bmod p$
    $c_1 = \mathsf{pad}(m) \cdot y^k \bmod p$
    return $(c_0, c_1)$
Function ElGamal_Decrypt($(c_0, c_1)$, $(g, p)$, $x$)
    $m' = c_1 \cdot c_0^{-x} \bmod p$
    $m = \mathsf{depad}(m')$
    return $m$
```

Vulnerable Implementations. Sequoia is safe since it does not support ElGamal encryption. All the other libraries are vulnerable, since their key validation steps are insufficient. Note that to run the attack, since ElGamal keys are encryption-only (namely they are necessarily subkeys) the adversary always needs to overwrite the primary key (to forge Certification Signatures). As a result, the adversary needs to deal with both primary and subkey key validation, but we now explain how neither prevent the attack. In OpenPGP.js and gopenpgp, key validation is only run over the primary key, which in the context of this attack can be converted into (or replaced by) a DSA one and then the corresponding (incomplete) key validation will always succeed by setting $g' = y' = 1$.[17]

In RNP and GPG, while both the primary key and the subkeys are validated, the adversary can always avoid the former validation by converting the key into a "GNU-dummy" one.[18] Further, the ElGamal key validation of both libraries does not perform enough checks to protect against the attack: the validation performed by RNP is always successful given our choice of public parameters, whereas GPG's ElGamal key validation can be bypassed by setting $g' = y' = 1$. Note that this does not affect the attack, because neither g nor y is used during ElGamal decryption.

The lack of sufficient key validation for ElGamal in all libraries is not surprising, since the order of the generator cannot be easily confirmed due to the fact that, following the OpenPGP specification, the group has composite order. Appendix D gives more details on what this entails in terms of key validation.

3.4.2 Cross-algorithm Attack against ECC keys. In the context of faulty signature attacks, we have seen how the DSA attacks could

[15] Set z at random mod p' and $g_t = z^{(p'-1)/2^t} \bmod p'$; such a g_t has order *dividing* 2^t; repeat the process until g_t has the desired order.

[16] Here, recall, we can assume that the combination of PKCS#1 v1.5 decoding and session key checksum ensure the oracle has no false negatives, so when $m' = \mathsf{pad}(m) \cdot g_1$, the oracle always indicates a decryption failure.

[17] If an RSA primary key with ElGamal subkey is targeted, meaning that the RSA key cannot directly be converted to a DSA one (due to incompatible secret parameters), the adversary can still convert the ElGamal (sub)key into a DSA one and overwrite both the primary and subkey with it.

[18] This is a non-standard key format (see https://dev.gnupg.org/source/gnupg/browse/master/doc/DETAILS$1497) which is supported by all the implementations in question. The GNU-dummy keys we consider include secret subkeys, but only the primary *public* key. The absence of secret parameters makes it impossible to run key validation.

be used to indirectly compromise ECC and ElGamal keys, since the private parameters have the same formats (see Section 3.3.1). The same trick applies here: the adversary can take any encrypted ECDH, EdDSA, ECDSA or DSA encrypted key packet, construct an ElGamal key packet from it, and then target it with the following decryption oracle attack. In the ECC setting, the bit-length of x is usually less than 521 (and 256 is common). So the ElGamal attack applied to ECC keys is very efficient in practice.

3.4.3 Decryption Oracle Attack against non-CRT RSA. RSA is vulnerable to a small subgroup attack, provided that the public modulus n is used in decryption (and not its factors). In such a case, the adversary can overwrite n as well as the public exponent e to recover d. OpenPGP again relies on PKCS#1 v1.5 encoding for RSA encryption, but unlike the ElGamal case, the padding needs to be handled for the RSA attack to be feasible.

We now explain how to pick n', e' and the ciphertext c so that the adversary can ultimately compute the private parameter d:

(1) Let $\mathsf{pad}(m)$ denote a PKCS#1 v1.5 encoded message of some bit-length k' (that need not be the same as the length of the original modulus n). Let p be small prime of the form $p = 2q + 1$ where q is also prime.[19] If $\mathsf{pad}(m)^q = 1 \bmod p$ and $\mathsf{pad}(m) \neq 1 \bmod p$, set $n' = \mathsf{pad}(m) \cdot p$ and overwrite the victim's n with n'. The value of e in the key does not matter as it is not used during RSA decryption.[20] If the conditions are not met, sample a different random padding in $\mathsf{pad}(m)$ and retry. This step will require on average 2 attempts (since the conditions we set only require $\mathsf{pad}(m)$ to be a quadratic non-residue modulo p). Thanks to our later exact choice of p, n' will be of a size such that the PKCS#1 v1.5 padding in $\mathsf{pad}(m)$ when taken modulo n' will include the correct leading bytes.

(2) For each $e' \in [1, q-1]$, query the decryption oracle on $c = \mathsf{pad}(m)^{e'} \bmod n'$; decryption will succeed if and only if $de' = 1 \bmod q$. When decryption succeeds, the adversary can recover $d \bmod q$ via $d = (e')^{-1} \bmod q$. If no value of e' works, then it must be that $d = 0 \bmod q$.

The adversary repeats the above process with different pairs p_i, q_i (the targeted $\mathsf{pad}(m)$ may also change) and combines the resulting $d \bmod q_i$ using the CRT to reconstruct d in full.

To see why the attack works, note that $\gcd(p, \mathsf{pad}(m)) = 1$ (by construction) hence using the CRT we have that $\mathsf{pad}(m)^{de'} = \mathsf{pad}(m) \bmod n'$ is equivalent to the pair of conditions

$$\mathsf{pad}(m)^{de'} = \mathsf{pad}(m) \bmod \mathsf{pad}(m)$$
$$\mathsf{pad}(m)^{de'} = \mathsf{pad}(m) \bmod p$$

The first condition always holds as both sides are equal to 0 mod $\mathsf{pad}(m)$, whereas the second condition holds if and only if $ed' = 1 \bmod q$, since $\mathsf{pad}(m)$ lies in the subgroup of $\mathbb{Z}_p^*$ of order q.

To ensure n' has the proper byte length such that the PKCS#1 v1.5 padding in $\mathsf{pad}(m) \bmod n'$ is correct, the adversary can pick a p having between 8 and 16 bits.[21] In terms of attack complexity, using a small p (and so a small q) lowers the total number of oracle queries needed to recover d in full, since in the worst case $q - 1$ decryption queries are needed to recover $d \bmod q$. While different values of n' are required to carry out the attack, note that the victim's key needs to be overwritten only once, as it suffices to add a different subkey for each n'. Hence, in some settings, picking fewer but larger primes q_i is preferable, depending on the maximum number of subkeys that are supported by the target implementation. Table 3 summarizes the cost of the attack for 2048-bit RSA.

	ElGamal (+ ECC)	RSA
Overwrites	1	1
Subkeys	1	234 for 2048-bit d
Decryptions	256 for 256-bit x 521 for 521-bit x	Worst case: $\sim$ 78k for 2048-bit d

Table 3: Number of key overwrites and oracle queries for each decryption attack, given common key sizes and smallest possible q_i (to minimize decryption queries).

Vulnerable Implementations. As already mentioned when discussing the RSA faulty signature attack, most implementations are protected since they rely on the CRT-RSA algorithm. OpenPGP.js is the only exception. If the primary key is an RSA one and key validation is performed, then the attack is not viable, unless the attacker can predict at which time the validation will be carried out (see Appendix C.2 for details on how to bypass the RSA validation process in OpenPGP.js). In all other cases, the library is vulnerable: as mentioned when discussing the ElGamal attack, encryption subkeys are never validated by OpenPGP.js, and if a DSA primary key is used, setting its public parameter to $g' = y' = 1$ always results in a successful validation.

3.4.4 Summary. We have presented a second class of KO attacks which takes advantage of decryption oracles to compromise any key, including signing ones. ECC and DSA private keys can be targeted by placing their encrypted key material inside an ElGamal key packet. The attack for ElGamal (and therefore for ECC and DSA) is particularly efficient in terms of the number of oracle queries required.

Exploiting decryption rather than signing can be convenient for interactive applications, especially if the adversary cannot acquire faulty signatures. The required oracle access might be obtained through error messages or timing leakage.

A summary of the cost of each attack in terms of oracle queries and key overwrites is given in Table 3, whereas Table 4 gives an overview of which libraries are potentially vulnerable based on their key validation methods (or lack thereof).

4 TARGETING OPENPGP APPLICATIONS

In this section, we examine the feasibility of the KO attacks presented previously in the context of some real-world applications. We first cover to what extent email clients might be vulnerable. Then, we consider two other deployment models that use potentially insecure/adversary-accessible storage. Specifically, we will

[19] p, q in this context have nothing to do with the secret RSA factors of a legitimate n.

[20] We assume that the RSA key being targeted has encrypt-only capabilities, hence e will not be used to verify a Primary Key Binding signature. If the original subkey can also be used for signing, then the adversary can easily disable that capability when forging the certification signatures.

[21] Larger p cannot be used since the PKCS#1 v1.5 encoding requires that $\mathsf{pad}(m)$ must have leading bytes `0x00`, `0x02`; smaller p would destroy the PKCS#1 v1.5 formatting.

	ElGamal (+ECC)	RSA
GPG	yes	no (CRT-RSA)
Sequoia	no (unsupported)	no (CRT-RSA)
RNP	yes	no (CRT-RSA)
OpenPGP.js	yes	depends
gopenpgp	yes	no (CRT-RSA)

Table 4: Vulnerability status of the different implementations with respect to decryption oracle key extraction attacks. If a library is not vulnerable, but not primarily due to key validation, then the reason is specified.

see how these vulnerabilities affected FlowCrypt and ProtonMail.[22] We discuss specific attacks against these particular applications to show how our attack ideas translate from isolated libraries to specific applications. Our aim is to give examples of vulnerable scenarios which could be found in other applications with similar features and authentication logic.

Targeting encrypt-to-self. In addition to the attacks detailed in Section 3, which exploit secret key operations (signing and decrypting), we briefly describe another attack vector which exploits encryption, and can be carried out against email applications, by taking advantage of the fact that emails are usually also encrypted to the sender's key, so that they can be stored in encrypted form and still be readable by the sender. The attack consists of replacing the public encryption key of the victim with one for which the adversary knows the private counterpart: any sent message will then be encrypted to the malicious key, and the adversary will be able to decrypt it, after intercepting the message. Compared to the key extraction attacks, one limitation of this encrypt-to-self compromise is that it only allows accessing sent messages, and not received ones (unless the sender quotes the received message when replying). Further, in the next paragraph we explain how secret key recovery has farther reaching consequences.

Impact of primary key recovery. In OpenPGP, compromising even a sign-only primary key could lead to long-term breach of confidentiality. In fact, if the adversary is able to obtain the primary private key of the victim, not only will they be able to sign messages with it (impersonating the victim), but they could also generate new key certification signatures. This allows an adversary to add encryption subkeys to the victim's key and advertise them publicly: as the third-party certifications will still be valid, the updated key will be considered legitimate by the victim's web-of-trust (see Section 2), and other users would likely start encrypting messages to the malicious encryption subkey. If the adversary were able to carry out a MITM attack, then they could intercept and access the messages, and re-encrypt them to the victim's original encryption subkey, to avoid detection.[23]

4.1 Email Clients

Email clients with OpenPGP integration usually support importing an existing key. Depending on where this key is stored by the user (before import), an adversary might have been able to overwrite it. Then, in the absence of key validation checks performed by the OpenPGP-extension in the client, if the user is not careful enough to inspect the key fingerprint, a corrupted key might be imported.

Decryption oracle attacks are quite challenging to mount in an email client setting. Such an attack might be possible if, for example, email decryption and subsequent rendering are automated by the email client. In such a case, the adversary can craft a message loading a specific image via URL (which they control), and determine whether a specific ciphertext decrypted successfully by monitoring accesses to the corresponding image URL. However, our attacks would require many such emails to be processed; in contrast the EFAIL attack [28] required only one. On the other hand, faulty signature attacks are viable. To obtain a faulty signature, the adversary might initiate an encrypted conversation with the victim: as OpenPGP messages are commonly signed, it is likely that the victim will include a signature in their reply message.

When it comes to attack detection, note that the adversary can corrupt the victim's key in such a way that the victim will still be able to decrypt older (and newer) emails that were encrypted to their legitimate key. In fact, the original victim's decryption subkey need not be removed when corrupting the key: the adversary only needs to replace its certification signature, so that it can be verified by the malicious primary key, thus imported and used by the client. Of course, the victim's past signatures will not verify anymore using the corrupted key, thus the sent emails will not be marked as authentic, however the emails in the inbox are signed by the sender. Hence, the victim might not immediately notice any issues with their imported key. Still, after obtaining the private key, in the general case the adversary cannot restore the original victim's key by replacing the imported malicious one, so the attack is likely to be detected eventually.

4.2 FlowCrypt

FlowCrypt provides a browser extension that allows users to take advantage of OpenPGP features when using Gmail. The application relies on the external email service (Gmail) to send, receive and store encrypted emails, which are actually processed by the trusted browser extension. Thus, the corresponding threat model is that of an untrusted server storing encrypted data. The browser extension directly communicates with the front-end code of the webmail service; the lack of interaction with a remote entity makes CCA attacks generally difficult, even though there is a still potential for in-browser attacks through e.g. cache attacks [10]. For instance, such an attack could be initiated by the Gmail web app.

Aside from the browser extension, FlowCrypt also offers mobile apps that connect to different email providers. To easily transfer the user's key across devices, as well as for backup purposes, FlowCrypt allows the user to store the encrypted private key in the user's inbox, by creating an email with subject "Your FlowCrypt Backup" and an attachment named "`flowcrypt-backup-<gmailaddress>.key`" that contains the encrypted key. When restoring a backed-up key from the inbox, the browser extension prompts the user to input the passphrase, but it does not show the corresponding key fingerprint. As long as the key decrypts successfully, it is imported. We now describe how the email provider, or an adversary with access to the

[22] FlowCrypt and ProtonMail developers were informed about the attacks. At the time of writing, the attacks described in this section are no longer possible.

[23] Since OpenPGP is vulnerable to surreptitious forwarding [5] unless the signature includes a signed Intended Recipient Fingerprint subpacket, the adversary could also relay any original message signature from the sender.

inbox, can use the backup email feature as an entry point for KO attacks.

4.2.1 Attacking FlowCrypt Users. A backed-up key can be imported directly into the FlowCrypt browser extension or mobile app. There are no key checks in place, neither when importing a key from a file nor when importing it from the inbox. As a result, an adversary who has access to the inbox can tamper with the backed-up key with the objective of carrying out a faulty signature attack to extract the private key, or an encrypt-to-self attack to read any sent messages, after the user has restored the malicious key from the backup. As a user is unlikely to restore a backup often, the two most viable attacks are the DSA and RSA ones that require to corrupt the key only once, and that can recover the secret from a single faulty signature (see the DSA Attack (variant 1) and Non-CRT-RSA attack in Section 3.3). Table 5 gives a summary of the feasibility of the various attacks we discussed. We verified the possibility of completing an RSA faulty-signature attack through backup email corruption by acting as the victim and simulating an attack against ourselves: we generated a fake back up email[24] containing a compromised RSA encrypted key (as per Section 3.3.4) and were able to import it successfully. We then sent out a signed email, whose embedded (faulty) signature allowed us to compute our original secret key (specifically, the secret exponent).

Attack Detection. The same comments for email client attacks also apply to FlowCrypt: the victim's key can be corrupted in a way that does not disrupt inbox decryption, however, new and old signatures in sent messages will not verify. Further, the adversary will probably be unable to hide their tracks by restoring the original key, unless the victim happens to re-import the backup key soon after the key extraction attack takes place.

4.3 ProtonMail

ProtonMail is built with an end-to-end encrypted architecture in mind, where the users do not have to trust the ProtonMail servers: sensitive data is encrypted by the clients and stored on the servers (this includes encrypted emails and keys). The users interact with either the ProtonMail web app, or the mobile applications (available for iOS and Android devices).

The code of the ProtonMail clients is open source, hence it can be inspected to check that sensitive information is not leaked to the server. However, for the web app, the code of the JavaScript front-end is downloaded from the server, which could send malicious code to target users. To prevent this issue and allow the web client to detect whether unexpected code was downloaded, a feature called "source code transparency" is planned [12]. Despite this protection not yet being in place, we assume that the JavaScript code used by the ProtonMail clients (including OpenPGP.js) has not been tampered with and matches the open-sourced code (otherwise attacks targeting users' private keys are trivial). An extensive overview of ProtonMail's architecture is given in the ProtonMail white paper [29]; the architecture was also independently reviewed in [18].

We now give a brief summary of the web app's logic. To login, the SRP-6a protocol [1] is used to confirm that the user knows the password, without sharing it with the server. If the login is successful, the user's encrypted keys are downloaded from the server, and the client tries to decrypt them with a passphrase which is derived from the user's password using bcrypt [30] with a unique salt. If the keys are already decrypted, or if they cannot be decrypted with the derived passphrase, a banner is displayed warning the user that the key could not be decrypted. If decryption is successful, then the keys are used to locally decrypt the user's mailbox, and generally trusted for any operation (including signing and encrypting-to-self new messages).

When sending a message, the client fetches the recipient's key from the server. In principle, a malicious server could perform a MITM attack and read the sent message by giving the wrong public key. To protect against this, users are able to mark a contact's key as trusted after having attested to its authenticity through other channels. Trusted keys are signed using the user's private key, and when composing a message to a trusted recipient, the user can visually check the validity of the fetched public key.[25] As the user composes a message, draft versions of it are periodically saved, signed and encrypted using the user's decrypted key and uploaded to the server: encryption to the recipients' keys is only performed after the user has clicked the "Send" button.

Cryptographic operations over confidential data are run client-side, however, some corresponding high-level errors may be reported to the server for diagnostic purposes. We will review how this might be problematic in specific cases.

Finally, aside from server-based attacks, when JavaScript code is run in the browser there is a risk that sensitive operations can be targeted using cross-tab cache attacks, as for FlowCrypt.

4.3.1 Attacking ProtonMail Users. ProtonMail applications rely on OpenPGP.js and gopenpgp, and they do invoke the corresponding key validation functions following successful key decryption. Still, due to the incompleteness of the key validation procedures, KO attacks are feasible. To target ECC or DSA keys, the main obstacle for the attacker (a malicious server) is getting hold of a DSA faulty signature. This can be achieved by waiting for the user to create or edit a contact (which is automatically signed and uploaded to the server). Another option is to run the faulty signature attack in combination with an encrypt-to-self attack, and wait for the user to compose a message, then extract the signature from the resulting draft. Targeting users with RSA keys[26] is more involved due to the key validation in place, and only possible in the web app. If the attacker is willing to expend the moderate effort required to factor 512-bit integers [33], they can bypass key validation for approximately 80% of logins. At a cost of only 30s of offline computation per login, the attacker can target 32% of logins. For details, see Appendix B.2. After bypassing key validation, an RSA faulty signature attack unfolds in the same way as for DSA.

As hinted, encrypt-to-self attacks are also possible, but they are as hard to mount as faulty signature attacks: key validation needs to be bypassed in the same way, and when the user is composing a

[24] To create a fake backup email that is correctly detected by FlowCrypt for import purposes, it sufficed to send an email with the same format as a legitimate backup email to our own Gmail address.

[25] See https://protonmail.com/support/knowledge-base/address-verification/.

[26] RSA was the default key type prior to July 2021, see https://github.com/ProtonMail/WebClients/commit/8ff22e1b190a88b4e726b20f4eb69c9c0e89a262.

	Faulty signature		Decryption oracle		Encrypt-to-self	
	RSA	DSA (+ECC)	RSA	DSA (+ECC)	RSA	DSA (+ECC)
FlowCrypt	yes	yes	no	no	yes	yes
ProtonMail	yes	yes	no	no	yes	yes

Table 5: Feasibility of the different types of key overwriting attacks in the context of specific applications, based on the primary key type.

message, then the automatically generated draft is always signed-then-encrypted. Hence, the best option for an attacker is to take advantage of these attacks not to directly break confidentiality, but as a means to obtain a faulty signature.

For decryption oracle attacks, while the possibility of user compromise is indeed present, concretely exploiting the vulnerabilities is likely infeasible, due to the lack of automated message decryption. In fact, emails are only decrypted when the user opens them, hence even though a malicious server could access the oracle by monitoring error messages sent to the server (for diagnostic purposes) when message decryption fails, recovering a significant number of bits of the private keys of ProtonMail users via this attack seems impractical. For an overview of the feasibility of all the different attacks against ProtonMail, see Table 5.

Attack Detection. In ProtonMail, the server has full control of which keys are sent to the user, meaning that the legitimate keys can be restored at the next login after carrying out a key extraction attack (or a partial encrypt-to-self attack). Potentially, compromising one user session is enough to complete a faulty signature attack. The only trace left behind would be some unverifiable sent messages or data, which the user could simply attribute to a bug.

5 COUNTERMEASURES

Short-Term Solution. To prevent the KO attacks presented here, implementations should not perform private key operations unless the integrity of the corresponding public parameters has been checked. This can be done by verifying that the correct relationship holds between the private key and the public key parameters (since the private key is integrity protected through the hash-then-encrypt mechanism used to lock the private key packet). Achieving this requires performing a series of key validation steps that depend on the specific algorithm; these are not specified as part of OpenPGP. Some implementations do already perform some form of key validation, but none of the libraries we have reviewed included sufficiently thorough checks to prevent all the attacks we have described. In Appendix D we detail the validation steps that need to be carried out to protect the keys of each algorithm type. However, these validation procedures, unlike the long-term solution discussed next, are not provably secure against e.g. potential variant KO attacks, hence they should only be used to safeguard legacy keys.

Long-Term Solution. Proper key validation is relatively expensive, algorithm-specific, and current implementations do not do it well. We believe that a better long-term solution is to use AEAD to guarantee full key integrity on decryption in an algorithm-agnostic way. AEAD-encrypted keys have already been added in the draft RFC 4880bis [19], and our proposal is a small extension of the existing specification that would not require any additional cryptographic functionality. All that is required is to include the public key material and the algorithm type in the Associated Data when creating AEAD-encrypted private keys. The standard security guarantees of AEAD ensure that the public key and algorithm type are then integrity-protected and bound to the encrypted private key. We worked with the OpenPGP working group on this AEAD proposal. It was added to the crypto-refresh document [20] in December 2021.

Finally, note that in implementations where the "encrypt-to-self" attack described in Section 4 is applicable, key decryption (and key validation for non-AEAD-encrypted keys) should be carried out before using the key for encryption, if the public parameters are not trusted.

6 CONCLUSIONS

In this paper, we have investigated key overwriting (KO) attacks against the OpenPGP specification, its implementations, and applications making use of those implementations. The attacks recover the private keys of users. Of particular note is how the changing landscape of real-world OpenPGP usage extends an attack avenue that dates as far back as 2001 [17].

We have seen how the lack of cryptographic binding between public parameters and (encrypted) private parameters in Secret Key packets allows cross-algorithm attacks in our setting, reducing the security of all discrete logarithm schemes to that of the weakest instance with interchangeable private key format. We also showed how the ability to overwrite public key parameters can be efficiently exploited for DSA and non-CRT RSA signatures, as well as ElGamal and non-CRT RSA decryption (assuming a single-bit side channel indicating decryption success or failure). Here we developed a range of novel attacks based on the particularities of the different algorithms, their implementations, and the non-standardised key validation procedures used in the main cryptographic libraries supporting OpenPGP. We found that some of the key validation procedures executed by specific libraries can themselves be exploited to carry out KOKV attacks. This highlights the danger of the OpenPGP specification leaving the task of confirming key integrity to individual implementations.

Finally, we proposed a simple, easy-to-deploy countermeasure to this class of attack: use an AEAD scheme to encrypt the private parameters of Secret Key packets, incorporating the public parameters (as well as the key algorithm) into the Associated Data. This countermeasure has now been included in the draft OpenPGP crypto-refresh document [20].

We also recommend deprecating the ElGamal encryption option in OpenPGP. As our work and [6] show, it is a dangerously weak encryption option, and proper key validation for it is expensive.

REFERENCES

[1] 2016 (accessed April 19, 2022). Improved Authentication for Email Encryption and Security. https://protonmail.com/blog/encrypted_email_authentication. (2016 (accessed April 19, 2022)).

[2] 2020 (accessed August 25, 2020). Stop Using Encrypted Email. https://latacora.singles/2020/02/19/stop-using-encrypted.html. (2020 (accessed August 25, 2020)).

[3] Martin R Albrecht, Jake Massimo, Kenneth G Paterson, and Juraj Somorovsky. 2018. Prime and prejudice: primality testing under adversarial conditions. In

Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 281–298.

[4] Daniel Bleichenbacher. 1998. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '98).* Springer-Verlag, Berlin, Heidelberg, 1–12.

[5] Don Davis. 2001. Defective Sign & Encrypt in S/MIME, PKCS# 7, MOSS, PEM, PGP, and XML. In *USENIX Annual Technical Conference, General Track.* 65–78.

[6] Luca De Feo, Bertram Poettering, and Alessandro Sorniotti. 2021. On the (In)Security of ElGamal in OpenPGP. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 2066–2080. https://doi.org/10.1145/3460120.3485257

[7] Hal Finney, Lutz Donnerhacke, Jon Callas, Rodney L. Thayer, and David Shaw. 2007. OpenPGP Message Format. RFC 4880. (Nov. 2007). https://doi.org/10.17487/RFC4880

[8] Hal Finney, Rodney L. Thayer, Lutz Donnerhacke, and Jon Callas. 1998. OpenPGP Message Format. RFC 2440. (Nov. 1998). https://doi.org/10.17487/RFC2440

[9] Steven D. Galbraith, Jake Massimo, and Kenneth G. Paterson. 2019. Safety in Numbers: On the Need for Robust Diffie-Hellman Parameter Validation. In *Public-Key Cryptography - PKC 2019 - 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II (Lecture Notes in Computer Science)*, Dongdai Lin and Kazue Sako (Eds.), Vol. 11443. Springer, 379–407. https://doi.org/10.1007/978-3-030-17259-6_13

[10] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. 2018. Drive-by key-extraction cache attacks from portable code. In *International Conference on Applied Cryptography and Network Security.* Springer, 83–102.

[11] Matthew Green. 2014 (accessed August 25, 2020). What's the matter with PGP? https://blog.cryptographyengineering.com/2014/08/13/whats-matter-with-pgp/. (2014 (accessed August 25, 2020)).

[12] Daniel Huigens. 2020 (accessed August 31, 2020). Building a Web App that Doesn't Trust the Server. https://archive.fosdem.org/2020/schedule/event/dip_securing_protonmail/. (2020 (accessed August 31, 2020)).

[13] Kahil Jallad, Jonathan Katz, and Bruce Schneier. 2002. Implementation of chosen-ciphertext attacks against PGP and GnuPG. In *International Conference on Information Security.* Springer, 90–101.

[14] Andrey Jivsov. 2012. Elliptic Curve Cryptography (ECC) in OpenPGP. RFC 6637. (June 2012). https://doi.org/10.17487/RFC6637

[15] Burt Kaliski. 1998. PKCS #1: RSA Encryption Version 1.5. RFC 2313. (March 1998). https://doi.org/10.17487/RFC2313

[16] Jonathan Katz and Bruce Schneier. 2000. A Chosen Ciphertext Attack against Several E-Mail Encryption Protocols. In *Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9 (SSYM'00).* USENIX Association, USA, 18.

[17] Vlastimil Klíma and Tomas Rosa. 2002. Attack on Private Signature Keys of the OpenPGP Format, PGP (TM) Programs and Other Applications Compatible with OpenPGP. *IACR Cryptol. ePrint Arch.* 2002 (2002), 76.

[18] Nadim Kobeissi. 2018. An Analysis of the ProtonMail Cryptographic Architecture. *IACR Cryptol. ePrint Arch.* 2018 (2018), 1121.

[19] Werner Koch, brian m. carlson, Ronald Henry Tse, Derek Atkins, and Daniel Kahn Gillmor. 2020. *OpenPGP Message Format.* Internet-Draft draft-ietf-openpgp-rfc4880bis-10. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-openpgp-rfc4880bis-10 Work in Progress.

[20] Werner Koch and Paul Wouters. 2021. *OpenPGP Message Format.* Internet-Draft draft-ietf-openpgp-crypto-refresh-04. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-openpgp-crypto-refresh-04 Work in Progress.

[21] Moxie Marlinspike. 2015 (accessed August 25, 2020). GPG And Me. https://moxie.org/2015/02/24/gpg-and-me.html. (2015 (accessed August 25, 2020)).

[22] Jake Massimo and Kenneth G. Paterson. 2020. A Performant, Misuse-Resistant API for Primality Testing. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 195–210. https://doi.org/10.1145/3372297.3417264

[23] Florian Maury, Jean-Rene Reinhard, Olivier Levillain, and Henri Gilbert. 2015. Format Oracles on OpenPGP. In *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015.* San Francisco, United States, 220–236. https://doi.org/10.1007/978-3-319-16715-2_12

[24] Fabrizio Milo, Massimo Bernaschi, and Mauro Bisson. 2011. A fast, GPU based, dictionary attack to OpenPGP secret keyrings. *Journal of Systems and Software* 84, 12 (2011), 2088–2096.

[25] Serge Mister and Robert Zuccherato. 2005. An Attack on CFB Mode Encryption as Used by OpenPGP. In *Proceedings of the 12th International Conference on Selected Areas in Cryptography (SAC'05).* Springer-Verlag, Berlin, Heidelberg, 82–94. https://doi.org/10.1007/11693383_6

[26] Phong Nguyen. 2004. Can We Trust Cryptographic Software? Cryptographic Flaws in GNU Privacy Guard v1.2.3, Vol. 3027. 555–570. https://doi.org/10.1007/978-3-540-24676-3_33

[27] NIST. 2013. Digital Signature Standard (DSS). FIPS 186-4. (2013). https://doi.org/10.6028/NIST.FIPS.186-4

[28] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. 2018. Efail: Breaking S/M IME and OpenPGP Email Encryption using Exfiltration Channels. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 549–566. https://www.usenix.org/conference/usenixsecurity18/presentation/poddebniak

[29] Proton Technologies A.G. 2016 (accessed August 31, 2020). ProtonMail Security Features and Infrastructure. https://protonmail.com/docs/business-whitepaper.pdf. (2016 (accessed August 31, 2020)).

[30] Niels Provos and David Mazieres. 1999. A Future-Adaptable Password Scheme.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 1999. 81–91.

[31] Scott Ruoti, Jeff Andersen, Daniel Zappala, and Kent Seamons. 2015. Why Johnny still, still can't encrypt: Evaluating the usability of a modern PGP client. *arXiv preprint arXiv:1510.08555* (2015).

[32] Steve Sheng, Levi Broderick, Colleen Alison Koranda, and Jeremy J Hyland. 2006. Why Johnny still can't encrypt: evaluating the usability of email encryption software. In *Symposium On Usable Privacy and Security.* ACM, 3–4.

[33] Luke Valenta, Shaanan Cohney, Alex Liao, Joshua Fried, Satya Bodduluri, and Nadia Heninger. 2016. Factoring as a Service. In *Financial Cryptography and Data Security - 20th International Conference, FC 2016, Christ Church, Barbados, February 22-26, 2016, Revised Selected Papers (Lecture Notes in Computer Science)*, Jens Grossklags and Bart Preneel (Eds.), Vol. 9603. Springer, 321–338. https://doi.org/10.1007/978-3-662-54970-4_19

[34] Filippo Valsorda. 2016 (accessed August 25, 2020). I'm giving up on PGP. https://blog.filippo.io/giving-up-on-long-term-pgp/. (2016 (accessed August 25, 2020)).

[35] Alma Whitten and J. Doug Tygar. 1999. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *Proceedings of the 8th USENIX Security Symposium, Washington, DC, USA, August 23-26, 1999*, G. Winfield Treese (Ed.). USENIX Association. https://www.usenix.org/conference/8th-usenix-security-symposium/why-johnny-cant-encrypt-usability-evaluation-pgp-50

A KEY VALIDATION STEPS IN IMPLEMENTATIONS

We detail the key checks implemented by each of the libraries we reviewed. As shown in Section 3 and Appendix C, these methods are often insufficient to prevent our attacks, and can actually introduce further vulnerabilities. For this reason, Appendix D discusses how to carry out proper key validation.

GPG. GPG validates keys when they are imported into the keystore, relying on the key checks implemented by libgcrypt.

DSA Given the public parameters g, p, y and private x, expect y to equal g^x mod p. See code at https://github.com/gpg/libgcrypt/blob/libgcrypt-1.9.3/cipher/dsa.c#L571.

ElGamal Same as DSA. See code at https://github.com/gpg/libgcrypt/blob/libgcrypt-1.9.3/cipher/elgamal.c#L473.

EdDSA No checks are done. See code at https://github.com/gpg/libgcrypt/blob/libgcrypt-1.9.3/cipher/ecc.c#L450.

gopenpgp. Gopenpgp (prior to v2.1.0) performs trial-signature validation by signing and immediately verifying a Literal Data packet using the primary key. The data being signed is not constant, but it is deterministic and it depends solely on the signature creation time. Keys are validated on key decryption. See code at https://github.com/ProtonMail/gopenpgp/blob/v2.0.1/crypto/key.go#L274.

OpenPGP.js. OpenPGP.js (prior to v4.10.5) uses the same validation strategy as gopenpgp. The only difference is that it does not use CRT-RSA for signing. See code at https://github.com/openpgpjs/openpgpjs/blob/v4.10.4/src/key/key.js#L405.

RNP. RNP (prior to v0.16) relies on the key checks implemented by Botan when the key is imported using the `rnp_import_keys` function, as done in the RNP CLI.[27] The checks are fairly extensive, however, when parsing secret keys, RNP re-derives some public key parameters, discarding the corresponding corrupted ones. This can help the attacker import corrupted keys in some cases, as we explain when relevant.

DSA Given the public parameters g, p, q and private x, Botan checks that g has order q, then verifies that $y = g^x \bmod p$ and that q is larger than x. However, RNP re-derives $y = g^x \bmod p$ beforehand, making the second check always pass, regardless of whether the attacker correctly guessed y'. See code at https://github.com/randombit/botan/blob/2.18.1/src/lib/pubkey/dsa/dsa.cpp#L59.

ElGamal Given the public parameters g, p and private x, Botan verifies that $y = g^x \bmod p$. Like with DSA, this check never fails as RNP sets $y = g^x \bmod p$ when loading the key. See code at https://github.com/randombit/botan/blob/2.18.1/src/lib/pubkey/elgamal/elgamal.cpp#L56.

EdDSA No checks are done, but they are also not needed as RNP re-derives the public point Q from the private parameter $seed$. See code at https://github.com/randombit/botan/blob/2.18.1/src/lib/pubkey/ed25519/ed25519_key.cpp#L96.

Sequoia. Sequoia does not carry out any key checks.

B FORGING KEY CERTIFICATION SIGNATURES AND BYPASSING RSA KEY VALIDATION

B.1 Forging Key Certification Signatures

For each of the algorithms that we target via key extraction attacks in Section 3, we now describe how to compute private parameters which can be used by the adversary to generate signatures that are verifiable given the maliciously chosen public parameters. This is necessary because the OpenPGP specification mandates that a key must not be used if its self-signature is missing or cannot be verified.

DSA When running our first attack, sample an arbitrary $s < q'$ coprime to q', then set $y' = 1$ and $r = [(g')^{hw} \bmod p']$. For the second attack, we can pick any $x' \in [2, q' - 1]$, set $y' = g'^{x'} \bmod p'$ and sign using g', x', p', q'.

EdDSA To carry out the faulty signature attack we can pick any point $Q' \neq Q$ on the curve. Hence to forge the binding signatures we are free to generate a fresh EdDSA keypair $(seed, Q')$, and sign using the private component.

RSA By construction we know $\phi(n')$, hence we can find $d' = e^{-1} \bmod \phi(n')$ and any message m can be signed as $s = m^{d'} \bmod n'$.

B.2 Bypassing RSA Key Validation in OpenPGP.js

Here we describe how the RSA key validation used by OpenPGP.js can be bypassed in a KO attack, at the same time as overwriting the key with a key (n', e') for which the private key d' is known. This enables key certification signatures to be forged, and thus allows mounting a KO attack on a signing or encryption subkey, through which the actual private key d can be recovered via faulty signature attack or decryption oracle attack, using the techniques in Sections 3.3.4 and 3.4.3. This combined attack is applicable to ProtonMail, as discussed in Section 4.3. The same attack vector can work against any application which relies on OpenPGP.js and which performs key validation at a time that is known/predictable for the attacker.

The attack uses similar ideas to the KOKV attack presented in Appendix C.2 (and the reader may find it helpful to read that appendix first).

The OpenPGP.js validation function creates an empty Literal Data packet and generates an OpenPGP signature over it. Let m be the data being signed: this includes the Literal Data packet payload, as well as additional metadata, like the signature creation time t. Let e, n be the public RSA values and d be the secret exponent in the primary key. The signature is computed over a PKCS#1 v1.5 padded message $\mathsf{pad}(m)$ of the form $\mathsf{pad}(m)$ = `0x00 || 0x01 || 0xFF ... 0xFF || 0x00 || hash_prefix ||` $\mathsf{hash}(m)$, where the hash function is SHA256 and the byte length of the signed message equals the public key size, i.e.$|\mathsf{pad}(m)| = |n|$. The signature is then $s = \mathsf{pad}(m)^d \bmod n$ which is verified by computing $s^e \bmod n$ and then checking that the resulting padded message $\mathsf{pad}(m)$ has the correct padding format and contains the correct message hash. If this verification succeeds, the key is considered valid.

We now show how to find n', e' to replace n, e such that key validation will always succeed. Let m_t be the message signed when key validation is performed at time t; the value of m_t can be predicted as a function of the signature creation time t (in seconds). We let $\mathsf{pad}(m_t)$ denote its PKCS#1 v1.5 encoding of length 512 bits (including the leading '0' bits). Given the padding format restrictions this is nearly the shortest valid padding that is possible. The adversary next factorises $\mathsf{pad}(m_t)$, aborting and trying the next m_t if this cannot be done quickly (we assess below how easy this factorisation is to carry out). The adversary then uses trial division to look for small factors of both $\mathsf{pad}(m_t) - 1$ and $\mathsf{pad}(m_t) + 1$, aborting and going back to factoring the next m_t if it does not find co-prime factors f_+ and f_- of $\mathsf{pad}(m) - 1$ and $\mathsf{pad}(m) + 1$, respectively, such that $f = f_+ \cdot f_-$ has between 8 and 16 bits. Once successful, the adversary sets $n' = \mathsf{pad}(m_t) \cdot f$ and sets e' to be any odd value, e.g. the commonly-used $e' = 65537$. Choosing n' in this way (with factor f of 8-16 bits in size) ensures that the selected encoding $\mathsf{pad}(m_t)$ is PKCS#1 v1.5 compliant.

We claim that, with this choice of (n', e'), message m_t will pass signature validation no matter the value of the original private key d. To see this, note that $\mathsf{pad}(m_t)^{de'} = \mathsf{pad}(m_t) = 0 \bmod \mathsf{pad}(m_t)$, while $\mathsf{pad}(m_t) = 1 \bmod f_+$ also implies $\mathsf{pad}(m_t)^{de'} = 1 = \mathsf{pad}(m_t) \bmod f_+$. Finally, $\mathsf{pad}(m_t) = -1 \bmod f_-$ implies $\mathsf{pad}(m_t)^{de'} = (-1)^{de'} = -1 = \mathsf{pad}(m_t) \bmod f_-$ (here we use the fact that e' is odd). Combining these three modular equations, we obtain:

$$\mathsf{pad}(m_t)^{de'} = \mathsf{pad}(m_t) \bmod n'.$$

This equation implies signature validation of m_t with key (n', e') will succeed.

[27] RNP also exposes the `rnp_load_keys` function, which carries out no validation. See code for both functions at https://github.com/rnpgp/rnp/blob/v0.15.1/src/lib/rnp.cpp

Now, since $n' = \mathsf{pad}(m_t) \cdot f$ and, by assumption, the adversary successfully factored $\mathsf{pad}(m_t)$, it can compute $\phi(n')$ to then find $d' = e'^{-1} \bmod \phi(n')$, which in turn allows creating a key certification signature that will verify using (n', e'). Note that, due to the non-standard form of n', only signatures created on values $\mathsf{pad}(m)$ that are co-prime to n' are guaranteed to successfully verify, but in the attack, the adversary has flexibility in choosing the message to be signed when creating the key certification signature.[28] Note also that the above computation can all be carried out offline, ahead of the actual attack.

As initially stated, the end goal of the attacker is not just to bypass key validation, but to be able to complete e.g. a faulty signature attack once the key is used for signing. To do so, after overwriting the primary key packet using (n', e'), it suffices to add a subkey that contains the original encrypted primary key material, and corrupt the public parameters as covered in Section 3. Whenever the victim signs using the corrupted key, the subkey will be used, as OpenPGP.js gives preference to signing using subkeys over use of the primary key.

Experimental results. To construct (n', e'), the attacker must be able to factor $pad(m_t)$ of size 512 bits. According to [33], 512-bit numbers are factorable with moderate computational effort in the worst case (where the number is a product of two primes): [33] reports a cost of \$75 and 4 hours of Amazon Elastic Compute time (2016 figures). So, in principle, with enough computational effort, every m_t can be used to try and build f.

However, it turns out that many numbers are much easier to factor than this. We used SAGE's built-in `factor` method, running on a MacBook Pro 2019. We generated 500 values of $pad(m_t)$ and set a time limit of only 30 seconds to factor each value. We were able to factor 195 (39%) of the candidate values. For 160 (32%) values we were then able to find suitable *primes* f_- and f_+ to build f.[29] We conclude that at least 32% of messages m_t can be used to mount the attack. Based on additional experiments (in which we just searched for suitable f_+, f_-), this figure could be increased to approximately 80% by putting more computational effort into the factoring step.

In the context of ProtonMail, for example, this means the adversary can target at least 1 in 3 user logins. We remark that $pad(m_t)$ is key- and user-independent, and the constructed e', n' can be used to target any user who has RSA keys and is logging into the web app at time t. Note that t is the user's local device time, and it is not controllable by ProtonMail, but it can still be predicted: the server could check for the client time skew based on e.g. diagnostic messages sent from the web app.

C KEY OVERWRITING ATTACKS EXPLOITING KEY VALIDATION (KOKV)

We have seen how some OpenPGP libraries carry out checks intended to validate key material before use. Such checks are not always sufficient to prevent the attacks we have described. Further, as we now show, some key validation steps open up new vulnerabilities in conjunction with key overwriting. We refer to these attacks as Key Overwriting attacks exploiting Key Validation (KOKV attacks). Since the key validation steps used are library specific (and not specified in the OpenPGP specification), all the discussion in this appendix is library-specific.

C.1 Exploiting ElGamal and DSA Key Validation to Recover the Private Key in GPG, OpenPGP.js and gopenpgp

Direct Key Validation. Whenever an implementation validates ElGamal or DSA keys by verifying that $y = g^x \bmod p$ without also checking the order of g, then the key validation function effectively acts as an oracle that can be exploited to carry out a key recovery attack by reusing the idea from Section 3.4.1 to extract private keys bit-by-bit.

Recall that, since we are still in the context of key overwriting attacks, the adversary controls the public values p, g, y (as well as q in the DSA case). The adversary begins by selecting a prime p' of the form $2^t h + 1$ and of the appropriate size. Here t is the bit-length of the exponent x, which is the target for our attack. We write $x = x_0 + x_1 \cdot 2 + x_2 \cdot 2^2 + \cdots x_{t-1} 2^{t-1}$. The adversary then finds g_t of order $2^t \bmod p'$. It sets $g_i = (g_t)^{2^{t-i}} \bmod p'$ so g_i has order $2^i \bmod p'$.

To recover bit x_0, the adversary overwrites p, g, y with p', g_1, y' where $y' = {g_1}^0 = 1$. Verification for these parameters succeeds if and only if $g_1^x = y' \bmod p$. Since g_1 has order 2, this condition is equivalent to ${g_1}^{x_0} = y' = g_1^0 \bmod p$. Hence key validation succeeds if and only $x_0 = 0$. So with the result of one key validation trial, the adversary recovers x_0.

To recover bit x_1, the adversary now overwrites p, g, y with p', g_2, y' where $y' = g_2^{x_0}$. Now key validation succeeds if and only if $g_2^x = y' = g_2^{x_0} \bmod p$. But g_2 has order 4, so this condition is equivalent to $g_2^{x_0+2x_1} = g_2^{x_0} \bmod p$, and we see that key validation succeeds if and only if $x_1 = 0$.

Proceeding in this way, replacing g with g_i and y with ${g_i}^{x_0+x_1 \cdot 2 + \dots x_{i-1} \cdot 2^{i-1}}$ at stage i, we can recover all the bits of x of bit-length t sequentially, using t key overwrites and t key validation queries.

Indirect Key Validation via Trial Signature. The DSA key verification mechanism implemented by OpenPGP.js and gopenpgp involves using the private key to generate a signature, and then attempting to verify the signature using the public key. The goal is again to confirm that the public parameter y corresponds to the secret x. However, this key validation approach can also be exploited to recover the private key. We use a classical small subgroup approach. The adversary can recover the partial secret $x \bmod q'$ for an arbitrary small prime q' as follows:

(1) Pick two primes p', q' such that q' is small and divides $p' - 1$ and find $g' \in \mathbb{Z}_{p'}^*$ with order q'.

(2) Select $x' \in [1, q']$ and set $y' = g'^{x'} \bmod p'$. Query the key validation oracle on (p', q', g', y'). If validation succeeds then we can infer that $x' = x \bmod q'$. Otherwise, sample the next x' and retry.

[28] In standard RSA, where $n = pq$ is a product of two primes, the co-primality condition is not needed, and any padded message can be signed. This no longer holds when, for example, n has a repeated prime factor, as can arise in our setting.

[29] Allowing f_- and f_+ to be composite (but still coprime) would give the attacker more flexibility, hence an even larger number of $pad(m_t)$ could be targeted; we did not implement this extension.

Let G be the group generated by the original g (in ElGamal, $|G| = p - 1$, whereas in DSA $|G| = q$). By running the steps above using different primes q'_i such that $\prod_i q'_i \geq |G|$, then the adversary finds different shares of the secret exponent $x \bmod q'_i$ which can be combined using the Chinese Reminder Theorem to get the full value of $x \bmod p$. The expected number of trials to find the correct x' in step 2 is $q'/2$. For this reason, the adversary wants to pick q'_i that are as small as possible. The simplest strategy is to select the primes sequentially starting from 2. For example, to recover a 2048-bit exponent, the adversary needs to use all primes up to 1481, which results in about 78k key validation queries. However, note that for 2048-bit ElGamal keys, GPG samples secret exponents that are only 338 bits long, while the secret scalars of ECC keys are always shorter than 521 bits. In these cases, it suffices to submit less than 7k queries on average. What this translates to in practice, in terms of attack cost, depends on the specific application, as we explain below.

Vulnerable Implementations. GPG, gopenpgp and OpenPGP.js expose the key validation oracle.

GPG (through libgcrypt) validates ElGamal and DSA keys by solely checking that $y = g^x \bmod p$, and is thus vulnerable to our first bit-by-bit key extraction attack for both key types.

OpenPGP.js and gopenpgp validate DSA primary keys via trial signatures. This allows an adversary to run the small subgroup attack provided above. OpenPGP.js and gopenpgp do not directly validate ElGamal keys since they are always encryption-only subkeys. Still, these keys can be targeted via cross-algorithm attack by converting them into DSA keys. EdDSA, ECDSA and ECDH keys can also be compromised in this way.

Practicality of the Attack. The feasibility of this DSA/ElGamal attack depends on whether the adversary can query the key validation oracle, and with what frequency. While a malicious ProtonMail server could have managed to carry out the attack in principle, it is unlikely that such an attack would go undetected, or could be carried out within a reasonable time frame. For more details, see Paragraph *Access to the Validation Oracle in ProtonMail* in Appendix C.2

Countermeasures. To avoid this class of attacks, validation must be performed as described in Section 5 and Appendix D.

C.2 Exploiting the RSA Key Validation in OpenPGP.js to Recover the Private Key

If textbook RSA (as opposed to CRT-RSA) is used for signing, then trial signature validation exposes an oracle that potentially allows to run a small subgroup attack to recover the RSA secret exponent. Signing using CRT-RSA avoids the attack as the RSA public parameters controlled by the adversary are not used during signing operations.

Recall that OpenPGP.js (prior to v4.10.5) relies on trial signature validation. We now detail how its RSA key validation can be exploited. The OpenPGP.js trial signature function creates an empty Literal Data packet and generates an OpenPGP signature over it. Let m be the data being signed, let e, n be the public RSA values and d be the secret exponent in the primary key. The signature is computed over a PKCS#1 v1.5 padded message $\mathsf{pad}(m)$ of the form $\mathsf{pad}(m)$ = `0x00 || 0x01 || 0xFF ... 0xFF || 0x00 || hash_prefix ||` $\mathsf{hash}(m)$, where the byte length of the signed message equals the public key size, i.e.$|\mathsf{pad}(m)| = |n|$. The signature is then $s = \mathsf{pad}(m)^d \bmod n$ which is verified by computing $s^e \bmod n$ and then checking that the resulting padded message $\mathsf{pad}(m)$ has the correct padding format and contains the correct message hash.

Let $\mathsf{pad}(m)$ have order $ord_n(\mathsf{pad}(m))$ in $\mathbb{Z}_n^*$. Then signature verification succeeds for this m if (and with high probability, only if) $ed = 1 \bmod ord_n(\mathsf{pad}(m))$, which means that the key is considered valid provided e is the inverse of d modulo the order of the signed message. In other words, the validation function acts an oracle that leaks whether the given e is the inverse of d modulo $ord_n(\mathsf{pad}(m))$. If $ord_n(\mathsf{pad}(m))$ is known and e and $ord_n(\mathsf{pad}(m))$ are co-prime, then through the Extended Euclidean Algorithm one can easily compute $d = e^{-1} \bmod ord_n(\mathsf{pad}(m))$. Note that for most m, we have $ord_n(\mathsf{pad}(m)) = \phi(n)$, and given a legitimate RSA public key, $\phi(n)$ is secret and hard to compute, hence the oracle cannot normally be exploited to find d given e.

However, we consider a KOKV adversary that can control the public parameters (n, e) being validated, and can also observe the outcome of the validation function (i.e. it can access the oracle described above). We also assume that the adversary knows the message $\mathsf{pad}(m)$ that will be signed – this is a reasonable assumption as we explain later. In this setting, the adversary can partially recover d using a modification of our RSA decryption attack from Section 3.4.3, as follows:

(1) Let p be a small prime of the form $p = 2q + 1$ where q is also prime.[30] Check that $\mathsf{pad}(m)^q = 1 \bmod p$. If this condition holds, set $n' = \mathsf{pad}(m) \cdot p$ where now $\mathsf{pad}(m)$ is a PKCS#1 v1.5 padding having an adversarially-selected bit-length k' (not necessarily the same as that of n). Otherwise, sample different p, q and retry. Thanks to our precise choice of p, n' will have the correct size so that the PKCS#1 v1.5 padding has the right format when reduced mod n'.

(2) For each $e' \in [1, q-1]$, submit (n', e') to the key validation oracle. Validation will succeed if and only if $de' = 1 \bmod q$. If no value of e' in the given range works, then it must be that $d = 0 \bmod q$; this can be determined after all $q - 1$ possible values of e' have been tried.

The adversary repeats the process with different p_i, q_i (the targeted $\mathsf{pad}(m)$ may also change) and combines the resulting $d \bmod q_i$ using CRT to reconstruct d in full.

It is possible to recover $d \bmod q$ through the oracle because by construction, $\gcd(p, \mathsf{pad}(m)) = 1$ hence, by the CRT, we have that $\mathsf{pad}(m)^{de'} = \mathsf{pad}(m) \bmod n'$ is equivalent to the pair of conditions

$$\mathsf{pad}(m)^{de'} = \mathsf{pad}(m) \bmod pad(m)$$

$$\mathsf{pad}(m)^{de'} = \mathsf{pad}(m) \bmod p$$

The first condition always holds as both sides are equal to 0 mod $\mathsf{pad}(m)$, whereas the second condition holds if and only if $de' = 1 \bmod q$, since $\mathsf{pad}(m)$ lies in the subgroup of $\mathbb{Z}_p^*$ of order q.

To end up with n' of proper byte length so that $\mathsf{pad}(m) \bmod n'$ is correctly padded, the adversary can pick p to be between roughly 8 and 16 bits in size. Using small p (and so small q) lowers the cost

[30] p, q in this context have nothing to do with the secret RSA factors of the original modulus n.

of each stage of the attack, since the average number of queries needed to recover $d \bmod q$ is $q/2$, but entails using more primes. For $q \sim 2^{b_q}$ and $n \sim 2^{b_n}$, the attack needs $\frac{b_n}{b_q}$ distinct primes. Assuming all the primes used have the same bit-size b_q (for simplicity), the total average number of queries is then $\approx 2^{b_q-1}\frac{b_n}{b_q}$. For example, with $b_n = 2048$ and $b_q = 13$, we get a cost for the full attack of approximately $2^{12} \cdot 2048/13 \approx 645$k queries. This can be reduced to 78k queries by using more carefully chosen primes of smaller bit sizes.

Practicality of the Attack. There are a number of things that make this attack feasible in OpenPGP.js: firstly, for key validation, the primary key is always used for signing, without first checking its self-signature. Secondly, the signed message is deterministic and it only depends on the signature creation time. Hence, if the time is predictable, so is $\mathsf{pad}(m)$ and the adversary can construct n' accordingly. In other words, if the adversary knows or guesses at what (client) time key validation will be run, then they can compute the corresponding $\mathsf{pad}(m)$ which will be used. In the context of the ProtonMail web app, for instance, the keys are validated immediately after being downloaded from the server, hence the latter can easily guess a set of possible $\mathsf{pad}(m)$ values that might be signed. However, in the specific case of ProtonMail, a malicious server is better off trying to bypass validation to mount a KO attack as covered in Appendix B.2 and Section 4.3, since running the KOKV attack just described is slower and more likely to be detected by the user, due to the need to compromise multiple login sessions.

D EFFECTIVE KEY VALIDATION STEPS

We detail the key checks to be carried out for each algorithm to protect against the key overwriting attacks in this paper. While these checks are sufficient for this purpose, some of them introduce a considerable performance overhead, hence we believe these should be used only as a short-term solution, until standardisation of a specification-level fix, like the one based on AEAD that we proposed in Section 5.

Some of the necessary checks can be carried out on the public parameters directly, whereas others involve secret parameters. Hence, in the latter case, key validation needs to be performed on a decrypted private key.

RSA Validation in RSA is not always necessary, depending on the specific signing/decryption algorithm used: if the public modulus n is used for signing or decrypting, then it's value should be re-derived from the secret factors p, q to verify that $n = pq$. This is not an expensive check, hence we would suggest performing it regardless of the RSA algorithm variant employed.

EdDSA The point Q in the public key can be re-derived from the secret seed, through scalar multiplication.

DSA DSA operations are secure provided the public group is strong, which is the case if the generator has large prime order. Simply checking that $y = g^x \bmod p$ is not sufficient to prevent all of our attacks, and it actually introduces an additional vulnerability, as discussed in Appendix C. Instead, for the parameters to be considered secure, the following checks should be made:

(1) p is prime
(2) q is prime[31] and larger than 160 bits
(3) q divides $p - 1$
(4) $1 < g < p$ and $g^q = 1 \bmod p$
(5) $y = g^x \bmod p$

Note that it is important that check (5) is performed last, to avoid enabling a small subgroup attack through the validation process, as described in Appendix C. Conveniently, the first 4 checks only involve public key parameters, hence they can be performed without decrypting the private key.

Recall that despite being a legacy algorithm, validating DSA keys is important as attacks against them can be used to indirectly compromise ECC keys.

ElGamal ElGamal validation is complicated by the fact that the OpenPGP standard does not mandate the use of a subgroup of prime order q – on the contrary, g may be generator of the full group, and thus have full (even) order $p - 1$. As a result, there is no efficient way to confirm the actual order of g. Still, we want to make sure that it is large enough to make small subgroup attacks infeasible. To achieve this, one solution, albeit expensive, is to check:

(1) $g^i \neq 1 \bmod p$ for all values $i \in [2, t]$ and large enough t
(2) $1 < g < p$ and $y = g^x \bmod p$

Again, these checks must be performed in the stated order. In step (1), the larger t is used, the more difficult a small subgroup attack becomes. For example, we can "force" the adversary to use primes of at least 16 bits in its attack by setting $t = 2^{16} - 1$, so that more than 500k queries would be required to recover a 256-bit secret. However, checking the order of g in this way is expensive, since every number up to a certain size must be checked in step (1).

As for DSA, ElGamal validation is needed to prevent attacks on ECC secret keys via a cross-algorithm attack. For reasons of efficiency and security, we believe that OpenPGP implementations should consider dropping support for ElGamal, and that the algorithm should be deprecated by the OpenPGP specification.

[31] For testing both p and q it is important to use a *robust* primality test since we are in the adversarial setting [3, 9, 22].