Caveat Implementor! Key Recovery Attacks on MEGA

Martin R. Albrecht[1], Miro Haller[2], Lenka Mareková[1], and Kenneth G. Paterson[2]

[1] Information Security Group, Royal Holloway, University of London
`{martin.albrecht,lenka.marekova.2018}@rhul.ac.uk`
[2] Applied Cryptography Group, ETH Zurich
`kenny.paterson@inf.ethz.ch, miro.haller@alumni.ethz.ch`

Abstract. MEGA is a large-scale cloud storage and communication platform that aims to provide end-to-end encryption for stored data. A recent analysis by Backendal, Haller and Paterson (IEEE S&P 2023) invalidated these security claims by presenting practical attacks against MEGA that could be mounted by the MEGA service provider. In response, the MEGA developers added lightweight sanity checks on the user RSA private keys used in MEGA, sufficient to prevent the previous attacks.

We analyse these new sanity checks and show how they themselves can be exploited to mount novel attacks on MEGA that recover a target user's RSA private key with only slightly higher attack complexity than the original attacks. We identify the presence of an ECB encryption oracle under a target user's master key in the MEGA system; this oracle provides our adversary with the ability to partially overwrite a target user's RSA private key with chosen data, a powerful capability that we use in our attacks. We then present two distinct types of attack, each type exploiting different error conditions arising in the sanity checks and in subsequent cryptographic processing during MEGA's user authentication procedure. The first type appears to be novel and exploits the manner in which the MEGA code handles modular inversion when recomputing $u = q^{-1} \bmod p$. The second can be viewed as a small subgroup attack (van Oorschot and Wiener, EUROCRYPT 1996, Lim and Lee, CRYPTO 1998). We prototype the attacks and show that they work in practice.

As a side contribution, we show how to improve the RSA key recovery attack of Backendal-Haller-Paterson against the unpatched version of MEGA to require only 2 logins instead of the original 512.

We conclude by discussing wider lessons about secure implementation of cryptography that our work surfaces.

1 Introduction

MEGA is a cloud storage and communication platform with over 265 million user accounts and more than 10 million daily users [MEG22a], advertising itself as secure and private by design. The platform distinguishes itself from other major providers by offering end-to-end encryption for stored data. On MEGA, user files should remain confidential even if the storage provider is malicious or has been compromised through a breach, implying security in a strong threat model. The security of MEGA in this setting was recently analysed in detail by [BHP23], which describes five attacks on the cryptographic protocol used by MEGA to authenticate users and encrypt user data. The first two of these attacks completely broke the confidentiality of user files. Shortly after, [RH22] significantly improved the first attack in [BHP23], reducing its requirement of 512 user logins to just 6.

At their heart, the attacks in [BHP23] exploit the lack of both key separation and integrity protection for stored keys in the MEGA design: a single user master key is used to encrypt both the user's RSA private key (used during user authentication) and the user's file encryption keys themselves; meanwhile AES in ECB mode is used for the encryption. This allowed the authors of [BHP23] to corrupt the RSA private key in certain ways that leaked useful information during the authentication protocol, as well as to "cut and paste" AES-ECB blocks from file encryption keys into the RSA private key.

The authors of [BHP23] proposed an immediate and non-invasive mitigation step in the form of adding a MAC to the existing construction.[3] In response, MEGA chose to not implement this or any of the other originally suggested countermeasures. Instead, MEGA added extra sanity checks in the client software to do more validation of payloads during or after decryption [MEG22d]. These checks were sufficient to prevent the specific attacks of [BHP23, RH22].

Shortly after MEGA released their patch addressing the attacks of [BHP23], they made one other change which (as we will show below) further increased the attack surface of their code: they added detailed error reporting during the decryption and sanity checking processes done by the client as part of the authentication protocol [MEG22e]. The errors produced during these steps are mostly distinguishable from one another and the error messages are sent to the server in place of the usual authentication response. A malicious storage provider can exploit this verbose behaviour, triggering the errors by supplying specially crafted inputs in an attempt to learn something about the decrypted data.

1.1 Contributions

In the MEGA infrastructure, each user has a master key $\mathtt{k_M}$ that is used with AES-ECB mode to encrypt multiple items, including the user's RSA private key and individual file encryption keys (in a special obfuscated format). In this work we describe two new attacks on the patched MEGA infrastructure in the malicious server setting which achieve an AES-ECB *decryption* capability under $\mathtt{k_M}$. These attacks can be used to recover individual 128-bit blocks of a target user's RSA private key. Combining this with lattice techniques, we can efficiently recover the entirety of the target user's RSA private key after recovering four specific blocks. Once this private key is recovered, the adversary can trivially decrypt the RSA ciphertexts appearing in file sharing messages to recover the keys needed to decrypt any files shared with the target user. The attacks can also be used to recover individual file encryption keys directly. These attacks continue to exploit the lack of key separation and integrity protection in the MEGA design, showing that the patch and further changes made by MEGA were not only insufficient but actively harmful.

Both attacks make use of an ECB *encryption* oracle that is present in the MEGAdrop feature, a part of the MEGA system that is supposed to be independent of the authentication protocol, yet uses the same master key $\mathtt{k_M}$. This feature enables the receiving of shared files from unregistered users. In short, MEGAdrop encrypts a newly shared file's encryption key to a user's public RSA key, but the user's client then silently re-encrypts that file encryption key under $\mathtt{k_M}$ using AES-ECB whenever the user is logged in. Since a malicious server can arbitrarily choose the file encryption key when sharing files with the user and then observe the resulting AES-ECB ciphertext, this provides the ECB encryption oracle that we need. For technical reasons explained later, we obtain two AES-ECB encrypted 128-bit blocks for each use of the oracle. Notably, the ECB encryption oracle can be realised without any user interaction.

The attacks also exploit the distinguishable errors arising during user authentication. We describe the individual errors in detail in Section 2. We also explain there how to instantiate the ECB encryption oracle. Both attacks can be seen as *key overwriting* attacks, since they rely on manipulating the values that are interpreted as the RSA private key by the client, and on including the target AES-ECB ciphertext block in a particular position in the encoded and encrypted RSA private key. This causes the errors that are triggered during client-side cryptographic processing to depend on the target plaintext block. User interaction is formally required for these attacks, which is why we measure their cost in terms of the number of login attempts they need (they are otherwise computationally inexpensive). As a secondary measure of attack complexity, we account for the number of ECB encryption oracle calls needed.

The first attack, described in Section 3, exploits an implicit error in the computation of modular inverses when sanity checking the RSA private key. It is an (un)fortunate consequence of an otherwise harmless bug in the code (not checking whether an inverse exists) which is caught by the client and

[3] This by itself does not suffice for authenticated encryption security, but presents the "immediate" level of countermeasures, i.e. the most easily achievable solution in the short term. [BHP23] outlines further levels of countermeasures termed "minimal" and "recommended", which provide better guarantees but require more fundamental changes to the MEGA platform.

reported to the server. The malicious server can use this oracle repeatedly to learn the value of the target AES-ECB plaintext block modulo a number of small primes, which enables recovery of the full block using the CRT. The attack requires on average $2^{9.29}$ login attempts per recovered AES-ECB plaintext block and 66 ECB encryption oracle queries per attacked user.

The second attack, described in Section 4, relies on how RSA decryption is carried out by the client during user authentication. It exploits a legacy artefact in the code that changes the resulting RSA plaintext length if a certain byte condition on the plaintext does not hold, in combination with an explicit error arising from a plaintext length check that is again reported to the server. The attack is a form of a small order subgroup attack [vW96, LL97], in that it relies on the ECB encryption oracle to overwrite the RSA primes p, q with values such that $(p-1)(q-1)$ has known small prime factors. In this way, the server can then force the client's decryption to take place in one of several small subgroups. The attack also overwrites the user's private key value d with a value that is completely known except in the target plaintext block. Then, the malicious server can use the length check oracle repeatedly to learn the value of d, and hence the target plaintext block, modulo each of the small primes. The final step again combines these values using the CRT to recover the target block.

We present two main versions of the second attack: one that is simpler but which requires a large amount of precomputation and one that is more complex but only requires negligible precomputation. On average, these versions require $2^{11.24}$ and $2^{11.63}$ login attempts per block, respectively. In both versions, this second attack requires a smaller number (up to 15) of ECB encryption oracle queries per attacked user than our first attack does. Further, this second attack exploits different errors from the first one and also relies on behaviours resulting from the "legacy" check on the second byte of plaintext. We include this attack to showcase that the existence of such checks and differentiated error reporting increases the attack surface.

Since the two attacks work on a per-block basis, we discuss how best to recover the entire RSA private key of the target user with the help of lattice techniques in Section 5. This reduces the number of blocks that need to be recovered using either of the two attacks to 4 instead of the 9 that would be required if the attacks were used directly to e.g. recover all of p. The attack complexity of recovering the full RSA private key using our first attack is then $2^{11.29}$ login attempts on average.

As a side contribution, we show in Section 6 how to combine the ECB encryption oracle obtained from the MEGAdrop feature with the second attack in [BHP23] to recover a target user's RSA private key from an *unpatched* MEGA client using only 2 logins (compared to the 512 logins needed in [BHP23] and the 6 needed in [RH22]). This shows that the original, unpatched MEGA system was even weaker than previously thought.

We conclude by briefly discussing attack mitigation in Section 7, noting the problematic nature of relying on easy-to-implement countermeasures that do not properly address the core security vulnerabilities. In that section, we also draw wider lessons from our work.

1.2 Related work

The work of [BHP23] provided a detailed overview of the MEGA infrastructure as well as attacks on confidentiality and integrity of user data stored on the platform. The follow-up work [RH22] significantly reduced the amount of user interaction required by the first attack of [BHP23] but was already prevented by MEGA's patches. The attacks in this work draw inspiration from the small-order subgroup attacks on DH [vW96, LL97] and the key overwriting attacks on OpenPGP [KR02, BPH22]. The use of a plaintext checking oracle is reminiscent of Bleichenbacher's attack on RSA with PKCS#1 v1.5 encoding [Ble98] but we target private key recovery rather than plaintext recovery.

1.3 Validation

We have verified the presence of the ECB encryption oracle, implemented the attacks and verified them in practice on test account data, using a MitM setup with mitmproxy [The22] to minimise interaction with the real MEGA servers and a locally-run MEGA web client (version 4.21.4) [MEG22b]. We made a single modification to the web client to automatically simulate repeated client login attempts after one initial manual login. The attacks were able to recover arbitrary AES-ECB-encrypted blocks of the test user's RSA private key with query costs consistent with our analysis (averaging $2^{9.30}$ login attempts for the first attack). We also implemented a proof of concept for recovering the entire RSA private key given four known blocks using lattice techniques.

1.4 Disclosure

We contacted MEGA to inform them of the vulnerabilities in their system on 29.09.2022. We suggested a 90-day disclosure period. We also suggested mitigations, stressing the importance of providing proper cryptographic integrity for data stored under users' master keys. MEGA acknowledged receipt of our disclosure on 30.09.2022. They said they would begin working on fixes and liaise with us before deploying them. We expect the disclosure process to be complete and the vulnerabilities to be mitigated by the conclusion of the Eurocrypt 2023 review process.

2 Oracles

2.1 Notation

We begin by establishing some notation that we use throughout.

Concatenation is denoted by $\|$. $[m]_k$ denotes an encryption of m under the key k, where the algorithm is determined by the context. B denotes bytes, and for x, $|x|_{\mathsf{B}}$ denotes the length of x in bytes and $|x|_{\mathsf{b}}$ denotes the length of x in bits.[4] For a tuple $X = (x_0, \dots, x_{n-1})$, $|X| = n$ denotes its size. For a byte string $\mathtt{m} = b_0 \| b_1 \| \dots \| b_{n-1}$ of length n and $s, t \in \mathbb{N}$, we define $\mathtt{m}[s] := b_s$, and $\mathtt{m}[s : t] := b_s \| \dots \| b_{t-1}$ for $s < t$. An empty object is denoted by $\mathtt{null}$, and a zero byte by $\mathtt{00}$. $\mathsf{ZeroPad}(\mathtt{m}, n) := \mathtt{00} \| \mathtt{00} \| \dots \| \mathtt{m}$ such that $|\mathtt{m}|_{\mathsf{B}} = n$, i.e. left-pad $\mathtt{m}$ with zero bytes. If it is necessary to distinguish between a byte representation and other types, $\mathtt{m}$ (as opposed to m) denotes a byte string. Conversion between byte strings and integers remains implicit, so we may write $m \leftarrow \mathtt{m}$ and vice-versa. $(\mathbb{Z}/n\mathbb{Z})^\times$ denotes the multiplicative group of integers modulo n. By $x \leftarrow\$ S$ we denote x sampled uniformly at random from S. In our attacks, B denotes a target plaintext block, which is a byte string with $|B|_{\mathsf{b}} = 128$. To differentiate it from a value computed while attempting to recover this block (which could be different if the attack is not correct), we denote the computed value by B^*.

2.2 ECB encryption oracle

MEGA's webclient exposes an ECB encryption oracle under a user's master key $\mathtt{k_M}$. This oracle allows MEGA, or anyone controlling their infrastructure, to encrypt 32 bytes of chosen plaintext in AES-ECB mode under the target user's master key $\mathtt{k_M}$ in a single query. Since AES-ECB without any additional measures does not provide any integrity protection, ciphertexts containing blocks that the adversary queried to the oracle cannot be distinguished without additional tests on the expected structure of the plaintext.

The oracle stems from code related to the MEGAdrop feature. MEGAdrop enables anyone to upload files to a folder in the cloud storage of the recipient without needing an account on MEGA. The recipient activates MEGAdrop for one of their folders and obtains a link that they can share with others. Unlike shared folders, senders do not see any file stored in the MEGAdrop upload folder.

The left algorithm of Fig. 1 describes the upload feature of MEGAdrop. The adversary can pick some file key $\mathtt{k_F}$, nonce $\mathtt{N_F}$, file F, and attributes A during the upload of a file to the MEGAdrop folder at the link L. The upload feature locally encrypts the file with AES-CCM using $\mathtt{k_F}$ and some nonce $\mathtt{N_F}$ picked by the client. Backendal et al. describe MEGA's encryption in more detail on lines 2-11 of Fig. 2 in [BHP23].

To instantiate an ECB encryption oracle, the adversary sets $\mathtt{k_F^{obf}}$ to 32 bytes of its choosing. Since $\mathtt{k_F^{obf}} = (\mathtt{k_F} \oplus x) \| x$ for $x = \mathtt{N_F} \| \mathtt{T}_{cond}$, the obfuscated key defines the values for $\mathtt{k_F}$, $\mathtt{N_F}$, and $\mathtt{T}_{cond}$ used in the file encryption (cf. Fig. 4 in [BHP23]). The adversary can use the file reconstruction part of the framing attack described in [BHP23] to obtain a file F that, when encrypted with $\mathtt{k_F}$ and $\mathtt{N_F}$,

[4] For $x \in \mathbb{Z}$, the value of $|x|_{\mathsf{b}}$ as understood by the MEGA client implementations is not always exact. In the big integer representation used by the web client, $|x|_{\mathsf{b}}$ is normally rounded up to the closest multiple of 8 or 32.

produces the MAC tag value $\mathtt{T}_{cond}$. As a consequence, the adversary can pick some attributes A and run MEGAdrop.upload$(\mathtt{k_F}, \mathtt{N_F}, F, A, L)$. This uploads $\mathtt{k_F^{obf}}$, encrypted under the receiver's public RSA key, to the server.

Section 9.12 of MEGA's security white paper [MEG22c] states that to "conserve CPU cycles, RSA-encrypted keys are transformed into AES-encrypted keys when encountered". Indeed, the webclient regularly polls for new files in the background and, when encountering an RSA encrypted key $[\mathtt{k_F^{obf}}]_{pk}$, re-encrypts $\mathtt{k_F^{obf}}$ with $\mathtt{k_M}$ and AES-ECB to produce an AES-ECB ciphertext that we denote by $[\mathtt{k_F^{obf}}]_{\mathtt{k_M}}$. It then uploads this updated key to the server (cf. [MEG22k]) as shown in the right half of Fig. 1. Therefore, the malicious server can learn the AES-ECB plaintext-ciphertext pair $(\mathtt{k_F^{obf}}, [\mathtt{k_F^{obf}}]_{\mathtt{k_M}})$.

While testing this oracle in mitmproxy [The22], we noticed that the server can pretend that a new file was uploaded to a MEGAdrop folder. The webclient re-encrypts the key as described in Fig. 1 even if the recipient does not use MEGAdrop and the file has an invalid path. Thus, we have an efficient ECB encryption oracle that does not require any user interaction and leaves no persistent traces in the user's cloud storage. It encrypts 32 B per query and can be accessed repeatedly.

```
MEGAdrop.upload($\mathtt{k_F}, \mathtt{N_F}, F, A, L$)
1 : $[F]_{\mathtt{k_F}}, \mathtt{T}_{cond} \leftarrow$ File.enc($\mathtt{k_F}, \mathtt{N_F}, F$)
2 : $[A]_{\mathtt{k_F}} \leftarrow$ Attr.enc($\mathtt{k_F}, A$)
3 : $\mathtt{k_F^{obf}} \leftarrow$ ObfKey($\mathtt{k_F}, \mathtt{N_F}, \mathtt{T}_{cond}$)
4 : $pk \leftarrow$ Server.lookup($L$)
5 : $[\mathtt{k_F^{obf}}]_{pk} \leftarrow$ RSA.Enc($pk, \mathtt{k_F^{obf}}$)
6 : Server.upload($[F]_{\mathtt{k_F}}, [A]_{\mathtt{k_F}}, [\mathtt{k_F^{obf}}]_{pk}$)

Webclient.update()
1 : while true do
2 :   $\tau \leftarrow$ Server.fetch_update($\mathtt{k_M}, sk$)
3 :   if $\tau \neq \perp$ then
4 :     $[F]_{\mathtt{k_F}}, [A]_{\mathtt{k_F}}, [\mathtt{k_F^{obf}}]_{pk} \leftarrow \tau$
5 :     $\mathtt{k_F^{obf}} \leftarrow$ RSA.Dec($sk, [\mathtt{k_F^{obf}}]_{pk}$)
6 :     $[\mathtt{k_F^{obf}}]_{\mathtt{k_M}} \leftarrow$ AES-ECB.Enc($\mathtt{k_M}, \mathtt{k_F^{obf}}$)
7 :     Server.upload($[\mathtt{k_F^{obf}}]_{\mathtt{k_M}}$)
8 :   endif
9 : endwhile
```

Fig. 1. Pseudocode for the MEGAdrop feature from the perspective of the sender and recipient. The left function MEGAdrop.upload shows the encryption of a file F with key $\mathtt{k_F}$ and nonce $\mathtt{N_F}$, uploaded to the MEGAdrop folder with link L together with the encrypted file attributes A. The right function MEGAdrop.upload shows how active clients regularly poll for updates and re-encrypt node keys immediately if they are encrypted with RSA.

2.3 Oracles from decoding and decryption error reports

Consider the authentication and session ID exchange that takes place every time a user logs into their account, described in more detail in [BHP23]. Let $\mathtt{k_e}$ be the user's 128-bit symmetric encryption key derived from their password, $\mathtt{k_M}$ the user's 128-bit symmetric master key and (pk, sk) the user's 2048-bit RSA keypair.

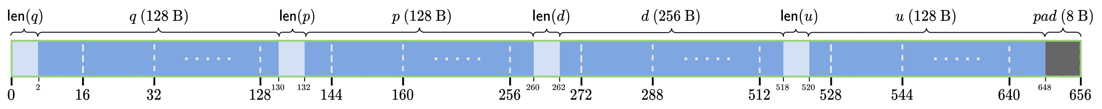

Fig. 2. Encoding of the RSA secret key together with the block boundaries marking the start of different 16-byte AES-ECB blocks. Each length encoding field consists of 2 bytes, meaning that data fields start progressively further into AES-ECB blocks.

Here, we focus on one part of this exchange, namely when the server responds to the user's request with the tuple $([\mathtt{k_M}]_{\mathtt{k_e}}, [\mathtt{privk}]_{\mathtt{k_M}}, [\mathtt{m}]_{pk}, \mathtt{uh})$, where $[\mathtt{k_M}]_{\mathtt{k_e}}$ and $[\mathtt{privk}]_{\mathtt{k_M}}$ are AES-ECB-encrypted, $[\mathtt{m}]_{pk}$

```
DecPrivkAndSid($\mathtt{k_M}$, $[\mathtt{privk}]_{\mathtt{k_M}}$, $[\mathtt{m}]_{pk}$, uh):
1 : require $|\mathtt{uh}|_B = 11$ else $\bot_1$
2 : privk ← AES-ECB.Dec($\mathtt{k_M}$, $[\mathtt{privk}]_{\mathtt{k_M}}$)
3 : $sk$ ← DecodePrivk(privk)
4 : m ← DecryptSid($sk$, $[\mathtt{m}]_{pk}$)
5 : require $|\mathtt{m}|_B = 255$ else $(\bot_2, |\mathtt{m}|_B)$
6 : require m[16 : 27] = uh else $\bot_3$
7 : sid ← m[0 : 43]
8 : return sid

DecryptSid($sk$, $[\mathtt{m}]_{pk}$):
1 : $N, e, d, p, q, d_p, d_q, u \leftarrow sk$
2 : $c \leftarrow [\mathtt{m}]_{pk}$
3 : require $c < N$ else $\bot_7$
4 : $x \leftarrow c^{d_p} \bmod p$; $y \leftarrow c^{d_q} \bmod q$
5 : $t \leftarrow x - y \bmod p$
6 : $h \leftarrow u \cdot t \bmod p$
7 : $m \leftarrow h \cdot q + y \bmod 2^{|N|_b}$
8 : m ← ZeroPad(m, $|N|_B$)
9 : if m[1] ≠ 00 then
10 :    m′ = 00 || m
11 : return m′[2 : $|\mathtt{m}'|_B$]

DecodePrivk(privk):
1 : $P$, pad ← Parse(privk)
2 : require $|P| = 4 \land |\mathtt{pad}|_B < 16$ else $\bot_4$
3 : $q, p, d, u \leftarrow P$
4 : $N \leftarrow p \cdot q$
5 : $e \leftarrow d^{-1} \bmod (p-1)(q-1)$
6 : $d_p \leftarrow d \bmod p$; $d_q \leftarrow d \bmod q$
7 : $u' \leftarrow q^{-1} \bmod p$
8 : require $u' \neq$ null else $\bot_5$
9 : $cond \leftarrow |p|_b, |q|_b, |u|_b > 1000 \land |d|_b > 2000$
10 : require $cond \land (u' = u)$ else $\bot_4$
11 : $sk \leftarrow N, e, d, p, q, d_p, d_q, u$
12 : require $e \neq$ null else $\bot_6$
13 : return $sk$
```

Fig. 3. Client decoding and decryption to process the session ID, derived from [MEG22n, MEG22l, MEG22m, MEG22j].

is RSA-encrypted and uh is in plaintext. Then, privk encodes the secret key *sk* for RSA-CRT as shown in Fig. 2, m encodes the session ID sid and uh is an 11-byte user handle string. The exact alignments of the fields in privk with respect to the AES-ECB block boundaries will be important in our attacks. The processing done by the client after it decrypts $[\mathtt{k_M}]_{\mathtt{k_e}}$ is shown in Fig. 3. This is the updated behaviour resulting from the patches described in Section 1 and converted into pseudocode as faithfully as possible, i.e. in some cases surfacing lower-level processing if it is relevant. In this section, we clarify further what is and what is not captured by our description.

In Fig. 3, we adopt the notation "**require** *condition* **else** *error*" to mean that the client checks the *condition* and if it is not satisfied, it aborts and outputs the *error* to the server. Decoding between base64-strings, bytes and integers is left implicit unless relevant to some error.

In DecodePrivk(privk), the function Parse(privk) sequentially reads through the bytes of privk whose expected form (as in Fig. 2) is

$$\mathsf{len}(\mathtt{q}) \,\|\, \mathtt{q} \,\|\, \mathsf{len}(\mathtt{p}) \,\|\, \mathtt{p} \,\|\, \mathsf{len}(\mathtt{d}) \,\|\, \mathtt{d} \,\|\, \mathsf{len}(\mathtt{u}) \,\|\, \mathtt{u} \,\|\, \mathtt{pad}$$

where len(x) denotes the two-byte big-endian length encoding of the byte-length of x and pad is padding, and returns the tuple of integers $P = (q, p, d, u)$. Computation of $a^{-1} \bmod b$ should be understood to return null if $\gcd(a, b) \neq 1$.

DecPrivkAndSid($\cdot$) contains a minor simplification: it is shown to directly operate on $[\mathtt{m}]_{pk}$, however in reality it receives $\mathtt{c} = \mathsf{len}([\mathtt{m}]_{pk}) \,\|\, [\mathtt{m}]_{pk}$. Note that if DecPrivkAndSid($\cdot$) returns successfully, then sid is revealed to the server in the requests that follow.

Notice that in addition to DecPrivkAndSid($\cdot$) returning a range of different error messages depending on the processing of secret values, it also modifies the resulting plaintext depending on whether the second byte of the RSA-decrypted value is `00` or not (line 9 of $\mathsf{DecryptSid}(\cdot,\cdot)$), a quirk that is explained in the original code only with the comment "Old bogus padding workaround" [MEG22j].

Caught and uncaught exceptions. Some of the errors shown in Fig. 3 are implicit, i.e. they are a result of lower-level exceptions caught at a higher level (the ones corresponding to $\perp_5$ and $\perp_6$). In all cases, they are shown at the exact place where the code aborts.

Further, due to some lower-level bugs in `asmcrypto.js` [MEG22f], the bigint and crypto library used by the web client, there are cases where the implementation never terminates:

- In $\mathsf{DecodePrivk}(\mathtt{privk})$ during $\mathsf{Parse}(\mathtt{privk})$, if one of q, p, d, u is 0.
- In $\mathsf{DecodePrivk}(\mathtt{privk})$ during the computation of $q^{-1} \bmod p$ [MEG22h], if $q \bmod p = 0$. We observed that this is because the implementation of $gcd(0, p)$ never terminates. The same issue arises during the computation of $d^{-1} \bmod (p-1)(q-1)$ if $d \bmod (p-1)(q-1) = 0$.

Similarly, there are cases when the implementation returns incorrect output:

- In $\mathsf{DecryptSid}(sk, c)$ during the computation of $x \leftarrow c^{d_p} \bmod p$ (and likewise $y \leftarrow c^{d_q} \bmod q$), there are several issues.
  - If p is even, the code computes $x = 0$ regardless of the other input values, because modular power computations were not implemented for even moduli [MEG22g].
  - If $|p|_{\mathsf{b}} > 1024$, the implementation of Montgomery reduction [MEG22i] does not return correct values, and so the output x is also incorrect.

We were forced to work around some of these implementation errors in our attacks.

3 Attack based on modular inverse computation

Our first attack enables block-by-block plaintext recovery of AES-ECB blocks encrypted under $\mathtt{k_M}$. In particular, this enables RSA private key recovery, i.e. the recovery of `privk`. Let $[B]_{\mathtt{k_M}}$ be such a target ciphertext block with unknown target plaintext block B, for example corresponding to an unknown block of q from `privk`.

This attack is in the malicious server setting, or equivalently the TLS-MitM setting, and makes use of the ECB encryption oracle described in Section 2.2. It exploits the error type $\perp_5$, which arises on line 7 and line 8 of $\mathsf{DecodePrivk}(\mathtt{privk})$ in Fig. 3 when $\gcd(p, q) \neq 1$. To get to this point, the server must submit inputs such that none of the previous error types are triggered. The server will only replace the $[\mathtt{privk}]_{\mathtt{k_M}}$ value and expect to abort before executing $\mathsf{DecryptSid}(\cdot,\cdot)$, so the only condition that must be satisfied is the one on line 2, which requires that the decrypted `privk` parses into 4 values without too much extra padding. Then, error $\perp_5$ can be distinguished from any of the errors that could follow, though with overwhelming probability this will be error $\perp_4$ from line 10 due to the server overwriting parts of `privk`.

The main idea behind this attack rests in the observation that if the server can construct $[\mathtt{privk}^*]_{\mathtt{k_M}}$ such that the decrypted and decoded p is divisible by a small prime r, and the decrypted and decoded q contains the target block B in its least-significant position, then the outputting of error $\perp_5$ leaks that $\gcd(p, q) \neq 1$ and thus (if some further conditions are satisfied), that $q \bmod r = 0$. From this, the server can learn the value of $B \bmod r$. Repeating this for a sufficient number of different primes r_i and combining the values using the Chinese Remainder Theorem (CRT), the server can learn the value of $B \bmod r_0 \cdot \ldots \cdot r_{n-1}$. If $|r_0 \cdot \ldots \cdot r_{n-1}|_{\mathsf{b}} \geq 128$, the server recovers B.

In the following subsections, we describe two versions of the attack in more detail, starting with the simple, block-aligned version and then describing an attack that is more general and resistant to simple fixes. Both versions have been implemented and verified using our MitM setup described in Section 1.3.

3.1 Block-aligned, small-length version

The attack proceeds in two distinct phases. The first phase calls the ECB encryption oracle to obtain a set of chosen-plaintext blocks, which are then combined with a target block to form the ciphertexts submitted to the client as part of the second phase. The second phase relies on the client making a number of online login attempts. The ECB encryption oracle calls are shown as $[\mathtt{x}]_{\mathtt{k_M}} \leftarrow \mathsf{OECB}_{\mathtt{k_M}}(\mathtt{x})$ (if $\mathtt{x}$ consists of 2ℓ blocks, this call will involve ℓ uses of the actual oracle described in Section 2.2).

Precomputation using the ECB encryption oracle. Take $\{r_0, \dots, r_{n-1}\} = \{7, 11, \dots, 103\}$, $n = 24$ small odd primes such that their product $R = \prod_{i=0}^{n-1} r_i$ has $|R|_{\mathsf{b}} \geq 128$. Let $[B]_{\mathtt{k_M}}$ be the target ciphertext block and denote by B^* the plaintext block computed as part of this attack.

1. Generate a random prime p' such that $|p'|_{\mathsf{b}} = 256$.
2. Let $d' \leftarrow 1$, $u' \leftarrow 1$ and encode them as byte strings $\mathtt{d}', \mathtt{u}'$ such that $|\mathtt{d}'|_{\mathsf{B}} = 254$, $|\mathtt{u}'|_{\mathsf{B}} = 126$.
3. Let $\mathtt{rest} \leftarrow \mathsf{len}(\mathtt{d}') \,\|\, \mathtt{d}' \,\|\, \mathsf{len}(\mathtt{u}') \,\|\, \mathtt{u}'$, which will have 4 non-zero blocks (2 of which are the same), and obtain $[\mathtt{rest}]_{\mathtt{k_M}} \leftarrow \mathsf{OECB}_{\mathtt{k_M}}(\mathtt{rest})$.
4. For $i \in \{0, \dots, n-1\}$, do the following:
   (a) Compute $p \leftarrow p' \cdot r_i$ and encode it as a byte string $\mathtt{p}$ such that $|\mathtt{p}|_{\mathsf{B}} = 126$.[5]
   (b) Let $\mathtt{ptp}_i \leftarrow \mathsf{len}(\mathtt{p}) \,\|\, \mathtt{p}$, which will be block-aligned and have 4 non-zero blocks. Obtain $[\mathtt{ptp}_i]_{\mathtt{k_M}} \leftarrow \mathsf{OECB}_{\mathtt{k_M}}(\mathtt{ptp}_i)$.
5. For $t \in \{0, \dots, r_{n-1} - 1\}$, do the following:
   (a) Compute $q^* \leftarrow 2^{128} \cdot t$ and encode it as a byte string $\mathtt{q}^*$ such that $|\mathtt{q}^*|_{\mathsf{B}} = 126$.
   (b) Let $\mathtt{ptq}_t \leftarrow \mathsf{len}(\mathtt{q}^*) \,\|\, \mathtt{q}^*[0:110]$, which will have 2 non-zero blocks and skips the last block of $\mathtt{q}^*$ to make space for the target. Obtain $[\mathtt{ptq}_t]_{\mathtt{k_M}} \leftarrow \mathsf{OECB}_{\mathtt{k_M}}(\mathtt{ptq}_t)$.
6. For $i \in \{0, \dots, n-1\}$, do the following:
   (a) For $t \in \{0, \dots, r_i - 1\}$, do the following:

      Store $\mathtt{ct}_{i,t} \leftarrow [\mathtt{ptq}_t]_{\mathtt{k_M}} \,\|\, [B]_{\mathtt{k_M}} \,\|\, [\mathtt{ptp}_i]_{\mathtt{k_M}} \,\|\, [\mathtt{rest}]_{\mathtt{k_M}}$. Notice that $\mathtt{ptp}_i[0:16]$ is the same for all i, and similarly $\mathtt{ptq}_t[0:16]$ is the same for all t.

Online attack. Suppose we have a set of $\mathtt{ct}_{i,t}$ as described above.

1. For $i \in \{0, \dots, n-1\}$, do the following:
   (a) For $t \in \{0, \dots, r_i - 1\}$, do the following:
      i. When the client initiates a login, respond to the client's request with $([\mathtt{k_M}]_{\mathtt{k_e}}, \mathtt{ct}_{i,t}, [\mathtt{m}]_{pk}, \mathtt{uh})$, where everything but $\mathtt{ct}_{i,t}$ is as it would be in an honest response.
      ii. If the client returns $\bot_5$, save the value of t and break out of this loop.
   (b) Save $B_i^* \leftarrow -2^{128} \cdot t \bmod r_i$.
2. Then, compute $B^* \bmod R$ by solving the system $B^* \equiv B_i^* \pmod{r_i}$ for $i \in \{0, \dots, n-1\}$ using CRT.

Why it works. Notice that for each decrypted $\mathtt{ct}_{i,t}$, $\mathsf{DecodePrivk}(\cdot)$ results in $p \leftarrow p' \cdot r_i$ and $q \leftarrow 2^{128} \cdot t + B$. The error $\bot_5$ will be triggered if and only if $\gcd(p,q) \neq 1$, which is equivalent to $\gcd(p,q) = r_i$, since p' is a prime larger than q. Hence $\bot_5$ is triggered if and only if $q \bmod r_i = 0$, and so if and only if $B \equiv -2^{128} \cdot t \pmod{r_i}$. This means that for the computed value B_i^* we have $B_i^* \equiv B \pmod{r_i}$. It follows that $B^* \equiv B \pmod{R}$. Since R is such that $|R|_{\mathsf{b}} \geq 128$ and $|B|_{\mathsf{b}} = 128$, we deduce that $B^* = B$ (over the integers).

[5] We include the prime p' for several reasons. First, because of one of the uncaught errors, we must make sure that $q \bmod p \neq 0$. Further, to avoid false positives from error $\bot_5$, we need the $\gcd(p,q) \neq 1$ signal to be equivalent to $\gcd(p,q) = r_i$.

Cost. To recover a block of plaintext, the attack requires $\frac{1}{2} \cdot (1 + 3 + n \cdot 4 + r_{n-1} \cdot 2) = 153 \approx 2^{7.26}$ uses of the ECB encryption oracle (recall that each use returns two blocks of ciphertext, and the all-zero plaintext block only needs to be queried once). With careful reuse of blocks across different key components (e.g. making use of the fact that the first block of $\texttt{ptp}_i$ is the same for all i), we can reduce the number of oracle queries to $\left\lceil \frac{1}{2} \cdot (1 + 3 + 2 + n \cdot 3 + r_{n-1}) \right\rceil = 91 \approx 2^{6.5}$. The results of these queries can be reused when recovering multiple blocks for a given target user. On average, the attack further requires $\frac{1}{2} \cdot \sum_{i=0}^{n-1} r_i = 627 \approx 2^{9.29}$ online login attempts (and $2^{10.29}$ in the worst case). Finally, the attack can be easily modified to use one less login for each r_i. This is because, in the online phase, if the server does not get a positive answer from the oracle for any of the values $t \in \{0, \ldots, r_i - 2\}$, it means that the value $r_i - 1$ is the correct one and so does not need to be submitted explicitly.

3.2 Full-length version

The attack in Section 3.1 could technically be prevented by a number of simple checks, e.g. by moving the check on bit lengths before the client computes $q^{-1} \bmod p$ (and so possibly triggers $\perp_5$), by ensuring that $|p|_\mathsf{b}, |q|_\mathsf{b} = 1024$ or that $d, u \neq 1$. However, none of these changes would prevent this type of attack: here we provide a more general version that would still work if these changes were made.

Precomputation using the ECB encryption oracle. As before, take $\{r_0, \ldots, r_{n-1}\} = \{7, 11, \ldots, 103\}$, $n = 24$ small odd primes such that their product $R = \prod_{i=0}^{n-1} r_i$ has $|R|_\mathsf{b} \geq 128$. Let $[B]_{\mathtt{k_M}}$ be the target ciphertext block and denote by B^* the plaintext block computed as part of this attack. Let $\texttt{ct} \leftarrow [\texttt{privk}]_{\mathtt{k_M}}$ be the original ciphertext encrypting the user's private RSA key.

1. Let $d' \leftarrow 2^{2047}$ and encode it as a byte string d' such that $|\mathsf{d}'|_\mathsf{B} = 256$.
2. Let $\texttt{ptd} \leftarrow \mathsf{00\ 00\ 00\ 01} \parallel \mathsf{len}(\mathsf{d}') \parallel \mathsf{d}'[0:10]$ and get $[\texttt{ptd}]_{\mathtt{k_M}} \leftarrow \mathsf{OECB}_{\mathtt{k_M}}(\texttt{ptd})$.
3. Let $[\texttt{rest}]_{\mathtt{k_M}} \leftarrow \texttt{ct}[272 : |\texttt{ct}|_\mathsf{B}]$. The slice taken from $\texttt{ct}$ begins with the ciphertext block that encrypts the most-significant full block of the original d.
4. For $i \in \{0, \ldots, n-1\}$, do the following:
   (a) Compute $p \leftarrow 2^{1023} + 2^{32} \cdot \varrho + 1$ for ϱ such that $p \equiv 0 \pmod{r_i}$ and p/r_i is prime. Encode it as a byte string p such that $|\mathsf{p}|_\mathsf{B} = 128$.
   (b) Let $\texttt{ptp}_i \leftarrow \mathsf{00\ 01} \parallel \mathsf{len}(\mathsf{p}) \parallel \mathsf{p}[0:124]$, which will likely have 2 non-zero blocks, and obtain $[\texttt{ptp}_i]_{\mathtt{k_M}} \leftarrow \mathsf{OECB}_{\mathtt{k_M}}(\texttt{ptp}_i)$.
5. For $t \in \{0, \ldots, r_{n-1} - 1\}$, do the following:
   (a) Compute $q^* \leftarrow 2^{1023} + 2^{128+16} \cdot t + 1$ and encode it as a byte string q^* such that $|\mathsf{q}^*|_\mathsf{B} = 128$.
   (b) Let $\texttt{ptq}_t \leftarrow \mathsf{len}(\mathsf{q}^*) \parallel \mathsf{q}^*[0:110]$, which will have 2 non-zero blocks, and obtain $[\texttt{ptq}_t]_{\mathtt{k_M}} \leftarrow \mathsf{OECB}_{\mathtt{k_M}}(\texttt{ptq}_t)$.
6. For $i \in \{0, \ldots, n-1\}$, do the following:
   (a) For $t \in \{0, \ldots, r_i - 1\}$, do the following:

       Store $\texttt{ct}_{i,t} \leftarrow [\texttt{ptq}_t]_{\mathtt{k_M}} \parallel [B]_{\mathtt{k_M}} \parallel [\texttt{ptp}_i]_{\mathtt{k_M}} \parallel [\texttt{ptd}]_{\mathtt{k_M}} \parallel [\texttt{rest}]_{\mathtt{k_M}}$. Notice that we have $\texttt{ptp}_i[0:2] = \mathsf{q}^*[126:128]$ for all q^* and $\texttt{ptd}[0:4] = \mathsf{p}[124:128]$ for all p. Further, notice that $\texttt{ptp}_i[0:16]$ is the same for all i, and similarly $\texttt{ptq}_t[0:16]$ is the same for all t.

Why it works. In this version, the precomputation must construct a modified ciphertext such that all values q, p, d, u are of the expected bit length. Recall that the plaintext encoding has the form: $\mathsf{len}(\mathsf{q}) \parallel \mathsf{q} \parallel \mathsf{len}(\mathsf{p}) \parallel \mathsf{p} \parallel \mathsf{len}(\mathsf{d}) \parallel \mathsf{d} \parallel \mathsf{len}(\mathsf{u}) \parallel \mathsf{u} \parallel \mathsf{pad}$. Since each value is encoded by prefixing a two-byte length field and the original lengths are either 1024 bits or 2048, the values in the resulting plaintext are not block-aligned. This is why we construct the "partial" block $\texttt{ptd}$ in Step 2 separately: it is composed of the final 4 bytes of p, $\mathsf{len}(\mathsf{d}')$ and the first 10 bytes of d'. Similarly, the block-aligned plaintext $\texttt{ptp}$ in Step 4b begins with another partial block which consists of the final 2 bytes of q^*,

$\mathsf{len}(\mathtt{p})$ and the first 12 bytes of p. Finally, the modified blocks are "stitched" together in Step 6a as in the simple version of the attack, ensuring that the target B is interpreted as the last "full" block of q.

Cost. Finding p of the correct form for each i in Step 4a is easy and takes $326 \approx 2^{8.35}$ trials on average for the given primes r_i. This step is independent of user data and so can be reused to attack multiple users. To recover a block of plaintext, the attack requires $\frac{1}{2} \cdot (1 + 1 + n \cdot 2 + r_{n-1} \cdot 2) = 128 = 2^7$ actual uses of the ECB encryption oracle. However, with more optimised reuse of blocks we can reduce the number of oracle queries to $\left\lceil \frac{1}{2} \cdot (1 + 1 + 2 + n + r_{n-1}) \right\rceil = 66 \approx 2^{6.04}$. As before, the results of the oracle queries can be reused when recovering multiple blocks from the same target user.

Online attack. Suppose we have a set of $\mathtt{ct}_{i,t}$ as described above.

1. For $i \in \{0, \ldots, n-1\}$, do the following:
   (a) For $t \in \{0, \ldots, r_i - 1\}$, do the following:
      i. When the client initiates a login, respond to the client's request with $([\mathtt{k_M}]_{\mathtt{k_e}}, \mathtt{ct}_{i,t}, [\mathtt{m}]_{pk}, \mathtt{uh})$, where everything but $\mathtt{ct}_{i,t}$ is as it would be in an honest response.
      ii. If the client returns $\perp_5$, save the value of t and break out of this loop.
   (b) Save $B_i^* \leftarrow (2^{16})^{-1} \cdot (-2^{1023} - 2^{128+16} \cdot t - 1) \bmod r_i$.
2. Then, compute $B^* \bmod R$ by solving the system $B^* \equiv B_i^* \pmod{r_i}$ for $i \in \{0, \ldots, n-1\}$ using CRT.

Why it works. Recall that for each decrypted $\mathtt{ct}_{i,t}$, $\mathsf{DecodePrivk}(\cdot)$ gets $p \leftarrow 2^{1023} + 2^{32} \cdot \varrho + 1$ and $q \leftarrow 2^{1023} + 2^{128+16} \cdot t + 2^{16} \cdot B + 1$. The overwritten values are encoded so that the parsing succeeds, and there are no other explicit errors that could be triggered before the error we are using for the attack.[6] The error $\perp_5$ will be triggered if and only if $\gcd(p, q) \neq 1$, which is equivalent to $\gcd(p, q) = r_i$ with high probability, since p/r_i is a large prime and the probability that $q \equiv 0 \pmod{(p/r_i)}$ is $\approx 1/(p/r_i) \leq 2^{-1016}$. Hence $\perp_5$ is triggered if and only if $q \bmod r_i = 0$, and hence if and only if $B = (2^{16})^{-1} \cdot (-2^{1023} - 2^{128+16} \cdot t - 1) \bmod r_i$. Thus we have $B_i^* \equiv B \pmod{r_i}$. The rest of the analysis follows as for the simpler version of the attack.

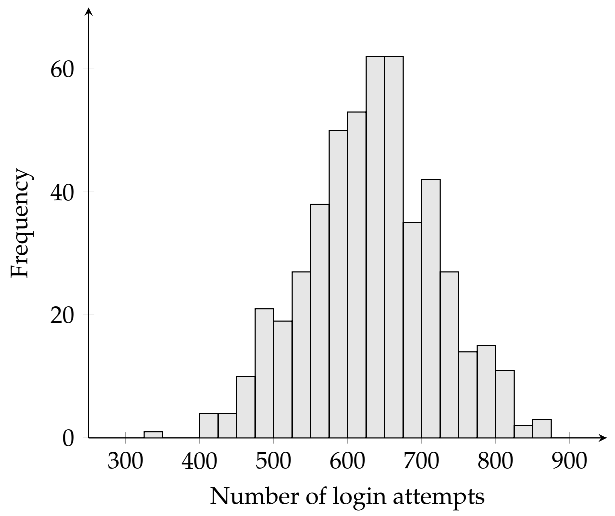

Fig. 4. Number of login attempts used by the attack over 500 runs.

[6] There is a possibility that $d^* \bmod (p-1)(q-1) = 0$ where $d^* \leftarrow d' + (d \bmod 2^{1968})$ and d is the original value encrypted in $\mathtt{ct}$. Because of the uncaught non-termination bug arising during the computation of $(d^*)^{-1} \bmod (p-1)(q-1)$, in this case the attack would fail, but this is highly unlikely to happen in practice.

Cost. The attack requires the same number of online login attempts as the simpler version in Section 3.1. We confirmed this in our implementation: in 500 runs of the attack recovering random ECB-encrypted blocks, the average number of login attempts required by the full version of the attack was $632 \approx 2^{9.30}$. The histogram is shown in Fig. 4.

4 Attack based on small subgroups

Here, we present our second AES-ECB decryption attack. In terms of login attempts it is less efficient than the attack in Section 3. However it requires fewer uses of the ECB encryption oracle. Further, it exploits a number of additional errors and also behaviours resulting from the "legacy" check on the second byte of the RSA plaintext.

The attack is also in the malicious server/TLS-MitM setting and uses the ECB encryption oracle from Section 2.2 with the aim of recovering blocks of d from the original privk (or any other AES-ECB-encrypted blocks that can be placed in their position). It exploits the errors $\perp_2$ and $\perp_3$ arising on line 5 and line 6 of $\mathsf{DecPrivkAndSid}(\mathtt{k_M}, [\mathtt{privk}^*]_{\mathtt{k_M}}, \mathtt{c}^*, \mathtt{uh}^*)$ in Fig. 3 for an adversarially supplied $\mathtt{privk}^*$ (created with the help of the ECB encryption oracle), $\mathtt{c}^*$ and $\mathtt{uh}^*$. It also requires working around some of the uncaught exceptions described in Section 2.3. To reach the needed error, the checks that trigger the earlier errors $\perp_1, \perp_4, \perp_5, \perp_6$ and $\perp_7$ must all be satisfied: $\mathtt{uh}^*$ must be a UTF-8 string of size 11, $\mathtt{privk}^*$ must encode q^*, p^*, d^*, u^* of sufficient length such that $\gcd(q^*, p^*) = 1$ and $\gcd(d^*, (p^* - 1)(q^* - 1)) = 1$ so that the corresponding inverses exist, $u^* = (q^*)^{-1} \bmod p^*$ and $c^* < N^*$ where $N^* = p^* \cdot q^*$.

Under these constraints, observe that $\mathsf{DecryptSid}(sk, \mathtt{c}^*)$ behaves differently depending on whether the second byte of the decrypted value $m^* \leftarrow (c^*)^{d^*} \bmod N^*$ is 00, where m^* is first zero-padded to the length of N^* to form $\mathtt{m}^*$. Suppose the server supplied p^*, q^* such that $|N^*|_{\mathsf{B}} = 256$. Let $\mathtt{m} \leftarrow \mathsf{DecryptSid}(sk, \mathtt{c}^*)$ and $\mathtt{m}'$ be the intermediate value such that $\mathtt{m} = \mathtt{m}'[2 : |\mathtt{m}'|_{\mathsf{B}}]$. Then, based on the error returned by the client, the server can distinguish the following two cases:

- Case $(\perp_2, 254)$: This means that $|\mathtt{m}|_{\mathsf{B}} = 254$, so $|\mathtt{m}'|_{\mathsf{B}} = 256 = |N^*|_{\mathsf{B}} = |\mathtt{m}^*|_{\mathsf{B}}$, so the condition on line 9 was not satisfied, i.e. $\mathtt{m}^*[1] = \mathtt{00}$.[7]
- Case $\perp_3$: This means that $|\mathtt{m}|_{\mathsf{B}} = 255$, so $|\mathtt{m}'|_{\mathsf{B}} = 257 = |N^*|_{\mathsf{B}} + 1 = |\mathtt{m}^*|_{\mathsf{B}} + 1$, which can only arise if $\mathtt{m}' = \mathtt{00} \parallel \mathtt{m}^*$ and so $\mathtt{m}^*[1] \neq \mathtt{00}$.

A similar case analysis can be done for arbitrary values of $|N^*|_{\mathsf{B}}$; then the errors may be swapped. However due to the bugs in the modular power implementation in MEGA code, the attack actually only works for $|N^*|_{\mathsf{B}} \leq 256$.

We explain next how to exploit this behavioural difference to leak information about a target user's RSA private key.

The server constructs $[\mathtt{privk}^*]_{\mathtt{k_M}}$ using the ECB encryption oracle such that in the "d" field it knows the plaintext for all blocks except the least-significant full block. That block will be the target of the attack; it can be an arbitrary AES-ECB-encrypted block $[B]_{\mathtt{k_M}}$. Let d^* denote the "d" component constructed in this way. The server must also precompute p^*, q^* of a special form and a number of values $\mathtt{m}^*$ with $\mathtt{m}^*[1] = \mathtt{00}$ such that it can interpret one of the errors arising on decryption of a corresponding ciphertext as confirmation of a correct "guess".

At a high level, the primes p^* and q^* are constructed so that $(p^* - 1)(q^* - 1)$ contains small prime factors r_i of a given bit length such that their product is at least 128 bits.[8] Let $\mathbb{G} = (\mathbb{Z}/N^*\mathbb{Z})^\times$ so that $|\mathbb{G}| = (p^* - 1)(q^* - 1)$. For each factor r_i, the server computes $g_i \in \mathbb{G}$ such that g_i has order r_i and such that a value $t_i \in \{1, \ldots, r_i - 1\}$ (or a set of such values $\mathcal{T}_i$) exists with the property that ${g_i}^{t_i} \bmod N^*$ has second byte 00 after zero-padding to the length of N^*. The value of u^* is then set to $(q^*)^{-1} \bmod p^*$.

[7] Note that the server does not know whether this is because prior to zero-padding, we have $|m^*|_{\mathsf{B}} \leq |N^*|_{\mathsf{B}} - 2$ and therefore trivially $\mathtt{m}^*[1] = \mathtt{00}$ or because $|m^*|_{\mathsf{B}} = |N^*|_{\mathsf{B}}$ and $\mathtt{m}^*[1] = \mathtt{00}$. However, the root cause is immaterial to our attack.

[8] The factors do not need to be common between $(p^* - 1)$ and $(q^* - 1)$, and can be freely distributed between the two.

Then, in the online phase of the attack, the server submits $\texttt{privk}^*$ constructed using the ECB encryption oracle to contain q^*, p^*, d^*, u^*. For each r_i, it sets $x_i = 1, 2, \ldots, r_i - 1$ and submits $c^*_{i,t} \leftarrow {g_i}^{x_i} \bmod N^*$ until the client returns the error that confirms the second byte of the decrypted value was 00 (which is $(\bot_2, 254)$ in the case that $|N^*|_\mathsf{B} = 256$ which we will use in the attack). Then, based on the precomputed values it learns that, for the specific value x_i triggering the error, $x_i \cdot d^* \equiv t_i \pmod{r_i}$. Here d^* is a value that is known except for its least significant full block, where it contains B. From this equation, the value of $B \pmod{r_i}$ can be recovered. Finally, using CRT and taking some care with non-block-aligned inputs, allows recovery of the block B.

The attack is described in more detail in the following subsections, first a simpler but less-efficient version and then the full version. The ECB encryption oracle calls are shown as $[\mathtt{x}]_{\mathtt{k_M}} \leftarrow \mathsf{OECB}_{\mathtt{k_M}}(\mathtt{x})$ as before. Since both versions of the attack must "stitch" AES-ECB blocks together to create the final ciphertext, we provide the algorithm in Fig. 5 to avoid repetition. This algorithm combines the chosen values $\mathtt{q}^*, \mathtt{p}^*, \mathtt{d}', \mathtt{u}^*$ so that they parse as expected, with the target block B being placed in the position of the least-significant full block of $\mathtt{d}^*$ and overwriting the corresponding block of $\mathtt{d}'$.

```
Stitch($\mathtt{q}^*, \mathtt{p}^*, \mathtt{d}', \mathtt{u}^*, [B]_{\mathtt{k_M}}$)
1 : $\mathtt{pt}_0 \leftarrow \mathsf{len}(\mathtt{q}^*) \parallel \mathtt{q}^* \parallel \mathsf{len}(\mathtt{p}^*) \parallel \mathtt{p}^* \parallel \mathsf{len}(\mathtt{d}') \parallel \mathtt{d}'[0:234]$
2 : $\mathtt{pad} \leftarrow\!\!\$\ (\{0,1\}^8)^8$   // random padding, could also be 00s
3 : $\mathtt{pt}_1 \leftarrow \mathtt{d}'[250:256] \parallel \mathsf{len}(\mathtt{u}^*) \parallel \mathtt{u}^* \parallel \mathtt{pad}$
4 : $[\mathtt{pt}_0]_{\mathtt{k_M}} \leftarrow \mathsf{OECB}_{\mathtt{k_M}}(\mathtt{pt}_0)$
5 : $[\mathtt{pt}_1]_{\mathtt{k_M}} \leftarrow \mathsf{OECB}_{\mathtt{k_M}}(\mathtt{pt}_1)$
6 : $\mathtt{ct}^* \leftarrow [\mathtt{pt}_0]_{\mathtt{k_M}} \parallel [B]_{\mathtt{k_M}} \parallel [\mathtt{pt}_1]_{\mathtt{k_M}}$
7 : return $\mathtt{ct}^*$
```

Fig. 5. Combining modified values produced using the ECB encryption oracle with the target ciphertext block in the correct format, reusing known AES-ECB blocks where possible. This assumes that $|\mathtt{q}^*|_\mathsf{B} = |\mathtt{p}^*|_\mathsf{B} = |\mathtt{u}^*|_\mathsf{B} = 128$ and $|\mathtt{d}'|_\mathsf{B} = 256$, as is the case for legitimate MEGA keys.

4.1 Simplified version

This version of the attack assumes a single t_i value per factor, which simplifies the presentation but imposes a high cost at the precomputation stage. Further, there is a non-negligible probability of the attack aborting and thus failing to complete. We will remove this restriction in the full version of the attack below.

Precomputation Take $\{r_0, \ldots, r_{n-1}\}$ where each r_i is a prime such that $|r_i|_\mathsf{b} = 8$, and n is such that $\left|\prod_{i=0}^{n-1} r_i\right|_\mathsf{b} \geq 128$. This imposes the constraint $16 \leq n \leq 19$. Let $[B]_{\mathtt{k_M}}$ be the target ciphertext block.

1. Find primes p^*, q^* such that $|p^*|_\mathsf{b} = |q^*|_\mathsf{b} = 1024$ and

$$p^* = 2 \cdot \left(\prod_{i=0}^{\lceil n/2 \rceil - 1} r_i\right) \cdot p' + 1, \quad q^* = 2 \cdot \left(\prod_{i=\lceil n/2 \rceil}^{n-1} r_i\right) \cdot q' + 1$$

   where p', q' is each a product of 2-4 large primes.[9] Encode p^*, q^* as byte strings $\mathtt{p}^*, \mathtt{q}^*$.
2. Set $N^* \leftarrow p^* \cdot q^*$ and $\mathbb{G} \leftarrow (\mathbb{Z}/N^*\mathbb{Z})^\times$.
3. For $i \in \{0, \ldots, n-1\}$:
   (a) Find $g_i \in \mathbb{G}$ of order r_i, e.g. by sampling $h \leftarrow\!\!\$\ \mathbb{G}$ and computing $g_i \leftarrow h^{(p^*-1)(q^*-1)/r_i} \bmod N^*$ until $g_i \neq 1$.

[9] These primes could repeat, the goal here is to avoid $(p^* - 1)(q^* - 1)$ having any other small factors except for $r_0, \ldots, r_{n-1}$.

(b) Find a value $t_i \in \{1, \ldots, r_i - 1\}$ such that for $m \leftarrow g_i^{t_i} \bmod N^*$; $\mathtt{m} \leftarrow \mathsf{ZeroPad}(m, N^*)$, we have $\mathtt{m}[1] = \mathtt{00}$. If no such t_i is found or there are multiple possible values, restart the precomputation.

4. Compute $u^* \leftarrow (q^*)^{-1} \bmod p^*$ and encode it as a byte string $\mathtt{u}^*$ with $|\mathtt{u}^*|_{\mathsf{b}} = 1024$.
5. Let $d' \leftarrow 2^{2047} + 1$ and encode it as a byte string $\mathtt{d}'$ with $|\mathtt{d}'|_{\mathsf{b}} = 2048$.
6. Obtain $\mathtt{ct}^* \leftarrow \mathsf{Stitch}(\mathtt{q}^*, \mathtt{p}^*, \mathtt{d}', \mathtt{u}^*, [B]_{\mathtt{k_M}})$. Let $d^* \leftarrow d' + 2^{48} \cdot B$ (where B is the unknown target block) denote the unknown value in the "d" field that will arise on decrypting $\mathtt{ct}^*$.[10]

Success probability. For random $m \in \mathbb{G}$ we have $\Pr[\mathtt{m}[1] = \mathtt{00}] = 2^{-8}$. For each factor r_i the probability that Step 3b finds exactly one suitable t_i is $(r_i - 1) \cdot 2^{-8} \cdot \left(1 - 2^{-8}\right)^{r_i - 2}$, which is greater than 0.18 for $2^7 < r_i < 2^8$. However, this needs to occur for all n factors where $n \geq 16$ to get a product of sufficient length to recover B using CRT, so the overall success probability is of the order $\approx 2^{-39}$ or less. To reduce the required amount of precomputation, in Section 4.2 we increase the bit length of each factor to ensure that there is at least one suitable t_i for each r_i and provide a strategy to disambiguate between multiple fitting t_i values.

Online attack Let $R = \prod_{i=0}^{n-1} r_i$ and $\mathtt{ct}^*, \{g_i\}_{i \in \mathcal{I}}, \{t_i\}_{i \in \mathcal{I}}$ be as computed before, for $\mathcal{I} = \{0, \ldots, n-1\}$.

1. When the client initiates a login, respond to the client's request with $([\mathtt{k_M}]_{\mathtt{k_e}}, \mathtt{ct}^*, [\mathtt{m}]_{pk}, \mathtt{uh})$, where everything but $\mathtt{ct}^*$ is as it would be in an honest response. If the client returns $\perp_6$, abort.
2. For $i \in \{0, \ldots, n-1\}$, do the following:
   (a) For $x \in \{1, \ldots, r_i - 1\}$, do the following:
      i. Compute $c_{i,x}^* \leftarrow (g_i)^x \bmod N^*$.
      ii. When the client initiates a login, respond to the client's request with $([\mathtt{k_M}]_{\mathtt{k_e}}, \mathtt{ct}^*, \mathtt{c}_{i,x}^*, \mathtt{uh})$, where everything but $\mathtt{ct}^*$ and $\mathtt{c}_{i,x}^*$ is as it would be in an honest response.[11]
      iii. If the client returns $(\perp_2, 254)$, save the value of x and break out of this loop.
   (b) If there is a saved value x, then we have $d^* \equiv x^{-1} \cdot t_i \pmod{r_i}$ for unknown d^*.
3. Then, use CRT to compute $d^* \bmod R$ from the values collected in Step 2b. Recall that by construction $d^* = d' + 2^{48} \cdot B$, so $d^* = 2^{2047} + 2^{48} \cdot B + 1$. Hence compute
$$B \equiv \left(2^{48}\right)^{-1} \cdot \left(d^* - 2^{2047} - 1\right) \pmod{R},$$
to recover the target plaintext block since $|R|_{\mathsf{b}} \geq 128$.

Cost. In the worst case, the main cost of the online attack is $\sum_{i \in \mathcal{I}} (r_i - 1)$ login attempts. This is bounded from above by $n \cdot (2^8 - 1) \approx 2^{12.24}$ for $n \leq 19$. In the average case, for each i we expect Step 2a to conclude after approximately $\frac{1}{2} \cdot 2^8$ trials, so the overall bound becomes $n \cdot 2^7 \approx 2^{11.24}$ for $n \leq 19$.

Probability of abort. Note that the attack aborts if it receives error $\perp_6$. This error is returned whenever the decrypted $d^* = d' + 2^{48} \cdot B$ is such that $\gcd(d^*, (p^* - 1)(q^* - 1)) \neq 1$. Since d^* is odd by construction,[12] the error can only be caused if at least one of the following is true:

[10] Note that by the choice of d', overwriting the least significant full block of d' with B is equivalent to adding $2^{48} \cdot B$ to d'.

[11] An honest response refers to the data that an honest server would have sent. Note that in this case, the "honest" $\mathtt{uh}$ will not match the value recovered from $\mathtt{c}_{i,x}^*$, but this check only comes after the errors triggered by the attack. The attacker could equally replace the $\mathtt{uh}$ value with an arbitrary 11-byte UTF-8 string.

[12] This is also why we cannot make the block-aligned simplification for this attack, because if we aligned it such that the least-significant block of d^* is full and therefore placed our target block B there, then if $B \equiv 0 \pmod{2}$ the client would output error $\perp_6$ on all queries.

- $d^* \equiv 0 \pmod{r_i}$ for at least one r_i,
- $d^* \equiv 0 \pmod{p'_j}$ for at least one $p'_j \mid p'$, or
- $d^* \equiv 0 \pmod{q'_k}$ for at least one $q'_k \mid q'$.

The values p'_j, q'_k are large primes by construction, so the probability of an abort being caused by those cases is negligible. However, each factor r_i is only 8 bits in size, which means that assuming a random B the probability that the attack aborts because $d^* \equiv 0 \pmod{r_i}$ for at least one r_i is bounded by $n \cdot 2^{-7} \approx 0.15$ with $n \leq 19$. In Section 4.2, we discuss strategies for avoiding the abort.

Why it works. Now, assume the attack does not abort. By construction, the values of q^*, p^*, d', u^* pass the check on bit length, we have $\gcd(q^*, p^*) = 1$, $u^* = (q^*)^{-1} \bmod p^*$ and all $c^*_{i,x} < N^*$. During $\mathsf{DecryptSid}(sk, \mathsf{c}^*_{i,x})$, the client will compute $m = \left(c^*_{i,x}\right)^{d^*} \bmod N^* = (g_i)^{x \cdot d^*} \bmod N^*$. If it is the case that $m = (g_i)^{t_i} \bmod N^*$ and therefore $x \cdot d^* \equiv t_i \pmod{r_i}$, the second byte of zero-padded m will be `00` and so the client will return $(\perp_2, 254)$ to the server. Otherwise, it will proceed with the computation and with very high probability return $\perp_3$, since the `uh` value will not match the relevant substring of m. Hence the attack recovers the target plaintext block.

4.2 Full version

Here, we provide strategies to improve the running time and the success probability of our second attack. First, we discuss the use of multiple t_i values per factor r_i, incorporate this into the attack and show the effect of this strategy. For practical purposes, this strategy is already sufficient to reduce the precomputation cost and the likelihood of aborts.

In this version of the attack, we increase the bit length of the factors r_i. As a result, the probability of finding a suitable t_i value during precomputation is increased. However this also implies that there will be more than one such value. We therefore have to also amend the online part of the attack to provide a way of determining which $t \in \mathcal{T}_i$ value has caused the expected error for a given x. There are multiple ways in which this could be achieved, and here we describe one option.

Take r_i, $\mathcal{T}_i$ and assume that we got the $(\perp_2, 254)$ error for some $x \in \{1, \dots, r_i - 1\}$. We can test each potential value $t_j \in \mathcal{T}_i$ by submitting another query $c^*_{i,x_j} \leftarrow (g_i)^{x_j} \bmod N^*$ where $x_j \leftarrow x \cdot t_j^{-1} \bmod r_i$. If the guess for t_j is correct, we have $x \cdot d^* \equiv t_j \pmod{r_i}$, and so decryption of c^*_{i,x_j} will produce $(g_i)^{x_j \cdot d^*} \bmod N^* = (g_i)^{x \cdot t_j^{-1} \cdot d^*} \bmod N^* = g_i$ as the plaintext. Then, as long as g_i is such that its second byte is not `00`, which we can ensure in the precomputation phase, the check that produces $\perp_2$ will pass. Since the server knows g_i and is able to set `uh` to arbitrary 11-byte values, it can also make sure to pass the check that produces $\perp_3$, and therefore get 43 bytes of g_i from the client via the returned `sid` value when the guess is correct. However, if the guess is not correct, it is very unlikely that the server-modified `uh` would match the resulting plaintext, leading to $\perp_3$. So the server can distinguish between the two cases.

Precomputation Take $\{r_0, \dots, r_{n-1}\}$ where each r_i is a prime such that $|r_i|_{\mathsf{b}} = 12$, and n is such that for $R \leftarrow \prod_{i=0}^{n-1} r_i$ we have $|R|_{\mathsf{b}} \geq 128$, so $11 \leq n \leq 12$. Let $[B]_{\mathtt{k_M}}$ be the target ciphertext block.

1. Find primes p^*, q^* such that $|p^*|_{\mathsf{b}} = |q^*|_{\mathsf{b}} = 1024$ and
$$p^* = 2 \cdot \left(\prod_{i=0}^{\lceil n/2 \rceil - 1} r_i \right) \cdot p' + 1, \quad q^* = 2 \cdot \left(\prod_{i=\lceil n/2 \rceil}^{n-1} r_i \right) \cdot q' + 1$$
where p', q' is each a product of 2-4 large primes. Encode p^*, q^* as byte strings $\mathtt{p}^*, \mathtt{q}^*$.
2. Set $N^* \leftarrow p^* \cdot q^*$ and $\mathbb{G} \leftarrow (\mathbb{Z}/N^*\mathbb{Z})^\times$.
3. For $i \in \{0, \dots, n-1\}$:

(a) Find $g \in \mathbb{G}$ of order r_i, e.g. by sampling $h \leftarrow\!\!\$\ \mathbb{G}$ and computing $g \leftarrow h^{(p^*-1)(q^*-1)/r_i} \bmod N^*$ until $g \neq 1$.

(b) Initialise $\mathcal{T}_i = \emptyset$.

(c) For $t \in \{1, \ldots, r_i - 1\}$, do the following:

i. Let $\mathtt{g}' \leftarrow \mathsf{ZeroPad}(g', N^*)$ for $g' \leftarrow g^t \bmod N^*$.

ii. If $\mathtt{g}'[1] = \mathtt{00}$, add t to $\mathcal{T}_i$. Else if $\mathtt{g}'[17 : \alpha]$ for some $\alpha \geq 28$ is a valid UTF-8 string of size 11[13], save $g_i \leftarrow g'$, $a \leftarrow t$ and $\mathtt{uh}_i^* \leftarrow \mathtt{g}'[17 : \alpha]$.

(d) If $\mathcal{T}_i = \emptyset$ or a is undefined, restart the precomputation.

(e) Shift $\mathcal{T}_i$ by replacing each $t \in \mathcal{T}_i$ by $t \cdot a^{-1} \bmod r_i$. This ensures that the values in $\mathcal{T}_i$ are with respect to the new generator g_i instead of g.

4. Compute $u^* \leftarrow (q^*)^{-1} \bmod p^*$ and encode it as a byte string $\mathtt{u}^*$ with $|\mathtt{u}^*|_\mathsf{b} = 1024$.
5. Compute $d' \leftarrow 2^{2047} + 2^{48+128} \cdot \delta + 1$ for $\delta < R$ such that $d' \equiv 0 \pmod{R}$. Encode it as a byte string $\mathtt{d}'$ with $|\mathtt{d}'|_\mathsf{b} = 2048$.
6. Obtain $\mathtt{ct}^* \leftarrow \mathsf{Stitch}(\mathtt{q}^*, \mathtt{p}^*, \mathtt{d}', \mathtt{u}^*, [B]_{\mathtt{k_M}})$.

Success probability. Increasing the bit length of the factors means that now for each factor r_i the probability that Step 3(c)ii finds at least one suitable t is $1 - (1 - 2^{-8})^{r_i - 1}$, which is greater than 0.9996 for $2^{11} < r_i < 2^{12}$. Across all n factors for $n \leq 12$, it is still greater than 0.99. Next, the probability that a random 11-byte string is a valid UTF-8 string is ≈ 0.001634. Hence for each factor r_i the probability that at least one such string will be found is $1 - (1 - 0.001634)^{r_i - 1} > 0.9648$, and across all factors it is at least 0.65. In practice, if the precomputation fails at this point, it can simply be re-run again with different r_i values.

Cost. This version tests all possible values of t for every r_i, so overall it must check at most $n \cdot 2^{12} \approx 2^{15}$ values of g^t (these can however be cycled through for each r_i). The prime generation is a one-time cost in the sense that the values can be reused in attacks on multiple users. Finally, since d' will be composed mostly of zero-blocks, building the ciphertext $\mathtt{ct}^*$ requires up to 15 uses of the ECB encryption oracle (which, recall, produces 2 blocks at a time).

Online attack Let $\mathtt{ct}^*, \{g_i\}_{i \in \mathcal{I}}, \{\mathcal{T}_i\}_{i \in \mathcal{I}}$ be the values computed before where $\mathcal{I} = \{0, \ldots, n-1\}$.

1. When the client initiates a login, respond to the client's request with $([\mathtt{k_M}]_{\mathtt{k_e}}, \mathtt{ct}^*, [\mathtt{m}]_{pk}, \mathtt{uh})$, where everything but $\mathtt{ct}^*$ is as it would be in an honest response. If the client returns $\perp_6$, abort.
2. For $i \in \mathcal{I}$, do the following:

(a) For $x \in \{2, \ldots, r_i - 1\}$, do the following:

i. Compute $c_{i,x}^* \leftarrow (g_i)^x \bmod N^*$.

ii. When the client initiates a login, respond to the client's request with $([\mathtt{k_M}]_{\mathtt{k_e}}, \mathtt{ct}^*, \mathtt{c}_{i,x}^*, \mathtt{uh})$, where everything but $\mathtt{ct}^*$ and $\mathtt{c}_{i,x}^*$ is as it would be in an honest response.

iii. If the client returns $(\perp_2, 254)$, save the value of x and break out of this loop.

(b) If $\mathcal{T}_i = \{t_i\}$ has a single element, skip this step. Otherwise, for $t \in \mathcal{T}_i$, do the following:

i. Let $x' \leftarrow x \cdot t^{-1} \bmod r_i$.

ii. Compute $c_{i,x'}^* \leftarrow (g_i)^{x'} \bmod N^*$.

iii. When the client initiates a login, respond to the client's request with $([\mathtt{k_M}]_{\mathtt{k_e}}, \mathtt{ct}^*, \mathtt{c}_{i,x'}^*, \mathtt{uh}_i^*)$, where only $[\mathtt{k_M}]_{\mathtt{k_e}}$ is as it would be in an honest response.

[13] Note that an 11 B byte string interpreted as a valid UTF-8 string will likely not be a string of size 11, i.e. a string consisting of 11 characters, since not all byte values are interpreted as text and non-ASCII characters require multiple bytes to encode [Wik22].

iv. If the client returns sid $= \mathsf{g}_i[1:44]$, save the value $t_i \leftarrow t$ and break out of this loop.

(c) We have that $d^* = d' + 2^{48} \cdot B \equiv x^{-1} \cdot t_i \pmod{r_i}$, and so $B \equiv \left(2^{48}\right)^{-1} \cdot x^{-1} \cdot t_i \pmod{r_i}$.

3. Then, use CRT to compute $B \bmod R$ from the values collected in Step 2c, which in turn recovers the target plaintext block since $|R|_{\mathsf{b}} \geq 128$.

Success probability. As in the attack in Section 4.1, this attack aborts if it receives error $\perp_6$. However, the probability that this happens becomes smaller with the increased bit length of the factors r_i. Assuming a random B, for 12-bit factors the probability of an abort is bounded by $n \cdot 2^{-11} \approx 0.006$ with $n \leq 12$. In Appendix A we give a more complex attack strategy that avoids the abort altogether.

Note on implementation. In practice, the attack's success probability may be impacted by another factor, namely differing implementations of UTF-8 validation. Suppose that the values g produced in Step 3(c)ii of the precomputation in Section 4.2 have valid UTF-8 substrings of size 11 in Python: this does not guarantee that they will be interpreted as such by the Javascript webclient. This requires implementing additional strategies for disambiguation in case the UTF-8-based one never yields the expected sid request. One alternative is to instead for all $t \in \mathcal{T}_i$ submit $x' \leftarrow x \cdot t^{-1} \cdot t_j \bmod r_i$ for some $t_j \in \mathcal{T}_i, t_j \neq t$, and use the original error $(\perp_2, 254)$ as the confirmation signal. This still has a potential for false positives and false negatives, however. A final, and most expensive, failover strategy is then to cycle through all values of x, saving the ones for which the client returns $(\perp_2, 254)$ and then running an offline computation to determine which x values are matched to which t values.

Cost. In the worst case, the main cost of the online phase of the attack is the $\sum_{i \in \mathcal{I}}(r_i - 1)$ login attempts needed. This is bounded by $n \cdot (2^{12} - 1) \approx 2^{15.58}$ for $n \leq 12$. In the average case, for each i we expect Step 2a to conclude after at most 2^8 trials and Step 2b to finish after around $\frac{1}{2} \cdot |\mathcal{T}_i| \approx \frac{1}{2} \cdot r_i \cdot 2^{-8}$ trials. Added together, the number of login attempts needed in the average case is bounded by $n \cdot (2^8 + \frac{1}{2} \cdot 2^{12} \cdot 2^{-8}) \approx 2^{11.63}$ for $n \leq 12$. Performing the experimental analysis over a large number of runs as in Section 3.2 would be more difficult due to the interaction between the disambiguation strategies and the web client with automated logins, which causes the web client to freeze or begin sending requests in large batches. This can impact the success rate (in particular, the attack may produce one x or t value that is slightly off) and hinders automating the attack. We stress that this is purely an artefact of our proof-of-concept implementation.

Note that to keep the presentation of the attacks simpler, we have assumed specific values of $|r_i|_{\mathsf{b}}$ and thus constrained the value of n. In reality, using different values would allow making a different tradeoff between the precomputation cost and the number of login attempts needed in the online phase. For instance, using 10-bit primes would lower the (online) worst-case bound to $n \cdot (2^{10} - 1) \approx 2^{13.91}$ for $n \leq 15$, but slightly increase the (online) average-case bound to $n \cdot (2^8 + \frac{1}{2} \cdot 2^{10} \cdot 2^{-8}) \approx 2^{11.92}$ login attempts. It would also make the precomputation phase much less likely to succeed in a single run: the probability of finding suitable t values for all r_i would fall to around 0.11, while the probability of finding generators with suitable UTF-8 substrings for all r_i would only be around 0.0002.

5 Recovering the RSA private key

Our attacks in Sections 3 and 4 can be seen as building generic AES-ECB decryption oracles. In this section, we turn this capability into an RSA private key recovery attack. Naively we would expect to call our costly AES-ECB decryption oracle up to nine times: each factor p, q of N has 1024 bits, but these are not perfectly aligned with AES block boundaries, necessitating to cover (partial) plaintexts from nine different 128-bit blocks. However, using a post-processing stage, we can reduce this number to four.

In particular, as illustrated in Fig. 2, the block alignments of p and q differ. For reasons that will become apparent below we will need to recover at least 512 bits. Based on the specific alignments, we will aim to recover the 512+16 least significant bits of q: 512 bits (i.e. four 128-bit blocks) are recovered using the attacks from Sections 3 and 4 and the least significant 16 bits are "recovered" using exhaustive search (which avoids the query cost of recovering a fifth block). If instead we targeted p, we would need to recover 32 bits using exhaustive search, which would have prohibitive cost. Thus, next, we discuss how to recover the remaining bits of q given the $\ell = 512 + 16$ least significant bits of q. In particular, we will solve the following computational problem.

Definition 1. *Let $N = p \cdot q$ be a 2048-bit RSA modulus with p, q having 1024 bits each. Given ℓ consecutive least significant bits of q, recover q.*

Our approach is a simple combination of exhaustive search, lattice reduction and root finding over $\mathbb{Z}$ following Coppersmith's method [Cop96]. In particular, we use the Howgrave-Graham variant [How97,How98,May10,MH20] of this algorithm. Let $\lceil \log_2 q \rceil - \ell < 1024$, $q = 2^{\lceil \log_2 q \rceil - \ell} \cdot r + q_0'$, where r are the bits we are trying to recover and $|q_0'| \leq 2^\ell$ are the known bits of q. Then r satisfies $f'(x) \equiv 0 \bmod q$ for $f'(x) := q_0' + 2^{\lceil \log_2 q \rceil - \ell} \cdot x \bmod q$. Given this we can consider

$$q_0 := 2^{-\lceil \log_2 q \rceil + \ell} \cdot q_0' \quad \text{and} \quad f(x) := q_0 + x \bmod q$$

and note that r still satisfies $f(x) \equiv 0 \bmod q$. That is, we translate our problem into one where the most significant bits are known rather than the least significant ones, cf. [MH20].

From this, the algorithm proceeds by constructing several polynomials that evaluate to zero modulo q or a multiple thereof, such as (powers of) N. In more detail, Let $h \geq 2 \in \mathbb{N}$ and $u < h \in \mathbb{N}$, for $0 \leq i < h$ we let

$$f_i(x) := \begin{cases} N^{u-i} \cdot (q_0 + x)^i & \text{for } 0 \leq i < u, \\ x^{i-u} \cdot (q_0 + x)^u & \text{for } u \leq i < h. \end{cases}$$

For example, picking $h = 4$ and $u = 2$ we get

$$N^2, \quad N \cdot q_0 + N \cdot x, \quad q_0^2 + 2\,q_0 \cdot x + x^2 \quad \text{and} \quad q_0^2 \cdot x + 2\,q_0 \cdot x^2 + x^3.$$

First, note that all $f_i(x)$ evaluate to zero modulo q^u at the correct r. Second, note the maximal degree of the $f_i(x)$ is $h - 1$, i.e. $\max_{0 \leq i < h}(\deg(f_i(x))) = h - 1$ and thus each polynomial has at most h coefficients.

Now, letting $X = 2^{\lceil \log_2 q \rceil - \ell}$ and $f_i^{(j)}$ denote the coefficient of x^j in $f_i(x)$, we construct a matrix $\mathbf{A}$ where the entry $A_{i,j} := f_i^{(j)} \cdot X^j$. Continuing with our example, we would have

$$\mathbf{A} := \begin{pmatrix} N^2 & 0 & 0 & 0 \\ N \cdot q_0 & N \cdot X & 0 & 0 \\ q_0^2 & 2\,q_0 \cdot X & X^2 & 0 \\ 0 & q_0^2 \cdot X & 2\,q_0 \cdot X^2 & X^3 \end{pmatrix}.$$

Since the matrix is triangular we can read off the determinant $\det(\mathbf{A}) = N^{u \cdot (u+1)/2} \cdot X^{h \cdot (h-1)/2}$. The rows of this matrix $\mathbf{A}$ span a lattice which contains a vector $\mathbf{v}$ of Euclidean norm $\|\mathbf{v}\| \leq \sqrt{h} \cdot \left(N^{u \cdot (u+1)/2} \cdot X^{h \cdot (h-1)/2}\right)^{1/h}$ by Minkowski's theorem. In other words, there exists an integer-linear combination of the rows of $\mathbf{A}$ that produces a vector with at most this Euclidean norm. Using lattice reduction we can find this shortest vector.[14] Now, given a vector of Euclidean norm $\|\mathbf{v}\|$ we know that its ℓ_1 norm, i.e. the sum of the absolute values of its entries, is bounded by $|\mathbf{v}|_1 \leq \sqrt{h} \cdot \|\mathbf{v}\|$. Finally, if $\mathbf{v} \neq 0$ and $|\mathbf{v}|_1 \leq q^u$, we can extract a polynomial that evaluates to zero modulo q^u on r but which evaluated at r is strictly smaller than q^u.[15] In other words, this polynomial evaluates to zero at r over $\mathbb{Z}$. The algorithm concludes by finding the roots of this polynomial, which can be accomplished in polynomial time (and efficiently in practice).

[14] The traditional presentation of this algorithm invokes the LLL algorithm which gives a short vector that is at most an exponential factor away from the shortest vector. However, the lattice dimensions involved here are well below the dimensions where the shortest vector problem (SVP) can be solved efficiently in practice – say, up to dimension 150 [DSvW21] – and we may thus simply assume we solve SVP. In any case, the exponential factor is $\approx 1.0219^h$ which is < 3 for $h \leq 50$.

[15] We extract $g(x)$ as $g^{(j)} := v_j / X^j \in \mathbb{Z}$.

To select h and u, by abuse of notation let h also be a formal variable and set $u := 1/2 \cdot h - 1$. As in [How98, p.102], we then find a root > 0 of

$$\frac{1024 - \ell}{2048} \cdot h \cdot (h-1) - u \cdot h + u \cdot (u+1).$$

This succeeds for $\ell > 512$ and the solution grows as ℓ approaches 512 from above.

As mentioned above, in our setting we consider $\ell = 512 + 16$: We run the attack from Section 3 on four blocks to recover 512 bits and run an exhaustive search over the remaining 16 bits (which are contained in a non-aligned block). In this setting, we picked $h = 36$ and $u = 18$. In our experiments, using LLL, finding a sufficiently short vector takes about 26 seconds on a Intel(R) Xeon(R) Gold 6252 CPU @ 2.10GHz using SageMath/FPLLL [S+22, The21]. In 1024 experiments, we obtained a success rate of 100%. Thus, we expect to be able to recover q in time $2^{16} \cdot 26$ seconds, or about 20 core days.[16] We give our proof of concept implementation in Appendix B and as an attachment.

The overall cost of the RSA private key recovery attack is $4 \cdot 2^{9.29} = 2^{11.29} \approx 2500$ login attempts, 66 ECB encryption oracle calls, and about 20 core days of computation (using the attack of Section 3 in combination with the attack in this section).

6 Attacking unpatched clients

We briefly revisit the attacks of [BHP23] against unpatched MEGA clients, as the presence of the ECB encryption oracle described in Section 2.2 has implications in this setting.

Attack 1 in [BHP23] uses an estimated 512 logins to recover a target user's RSA private key. The number of logins required was subsequently reduced to 6 in [RH22] by using more sophisticated lattice techniques.

Attack 2 in [BHP23] then exploits knowledge of that private key to recover two blocks of AES-ECB plaintext per login. This is done by overwriting two blocks of the encrypted version of u with the target AES-ECB ciphertext blocks and selecting a carefully crafted RSA ciphertext in the authentication protocol; the session ID returned by the client in that protocol then leaks the two AES-ECB plaintext blocks. This approach is used to build an efficient procedure for recovering file encryption keys in [BHP23].

Interestingly, however, the RSA private key used in Attack 2 in [BHP23] does not need to be the target user's true key – it only needs to be a key known to the adversary and any valid RSA private key (in the appropriate format) will do. Hence, an adversary can use the ECB encryption oracle to create a suitably encrypted, known RSA private key. By carefully reusing all-zero blocks for most of q, p and d, the number of ECB encryption oracle calls needed can be made as small as 7. The adversary then applies Attack 2 from [BHP23] with the target AES-ECB ciphertext blocks being selected from those encrypting the least significant bits of q (from the actual private key). With two applications of the attack, the adversary recovers 4 plaintext blocks, or 512 bits of q. Applying the lattice attack from Section 5, the adversary recovers the full RSA private key.

The cost of the attack is 2 login attempts and a small number of ECB encryption oracle calls.

Note that Attack 2 of [BHP23] is prevented by patched MEGA client code because of the requirement that the client-selected 11-byte string `uh` appear in `m` at a specific location and because overwriting u with a target ciphertext would make the check at line 10 in DecodePrivk(`privk`) fail.

[16] We note that this computation is "proudly parallel" or "embarrassingly parallel" because for each of our 2^{16} guesses we can run an independent lattice reduction. We also note that the running time is independent of whether the input instance corresponds to a correct or incorrect guess. Moreover, incorrect solutions resulting from incorrect guesses can be filtered out using the known public key.

7 Discussion and future work

On the one hand, the conclusion to be drawn from this work for practitioners and designers is no different from the one derived from [BHP23]. The root causes at play here were already identified in [BHP23], whose suggestion of protecting the integrity of encrypted keys using a MAC would have prevented the attacks in this work as well. Further, the existence of the ECB encryption oracle in a feature completely separate from the attacked protocol highlights the continued fragility of the MEGA infrastructure, made possible also by the lack of key separation.

However, our attacks also highlight issues going beyond the ones exposed in previous works. First, some of the errors that our attacks exploit as oracles are not explicit, but derive from bugs in the big integer arithmetic provided by `asmcrypto.js`. This presents a challenge already mentioned in [BBB+21] which called for a verified big integer library that could serve as a common core for different projects. In the case considered here, such a library would need to be cross-compilable to JavaScript or WebAssembly. We consider this a pressing area for future work.

There are also further lessons to be drawn for a cryptanalytic audience. First, our attacks serve as an additional example of key overwriting attacks [KR02, BPH22, BHP23], a class of attacks that appears to deserve more exploration in terms of targets (deployed in practice) and attack refinement. Moreover, our attacks make use of the detailed and verbose error reporting by MEGA clients. This enables powerful side-channel attacks that can be observed over the Internet[17], highlighting the practical significance of these classes of attacks. Finally, our work, along with other recent works attacking widely deployed protocols such as [VR20, LGR21, AMPS22, SRW22, BPH22], underlines that while it might seem that the "golden age" of cryptographic attacks against deployed protocols is over – given the level of academic involvement and formal rigour that went into the design of TLS 1.3 – the target has simply moved up the stack. As cryptographic applications move beyond "simple" protection of data in transit or at rest, more complex cryptographic solutions are deployed at scale, often without significant input from the cryptographic community. This suggests a broad and impactful field for cryptanalysis of targets "in the wild". It is well known that attacks are typically required to convince practitioners to adopt cryptographic recommendations. This in turn suggests that to achieve the adoption of more secure and formally analysed cryptographic solutions in practice, further cryptanalytical work on the "current generation" of deployed solutions is needed.

Finally, the two attacks presented in this work require a large number of login attempts. This was also the case for the first attack of [BHP23] and used as an argument by MEGA that the attack was not practical. However, later work by [RH22] reduced the number of login attempts to six, and we have further reduced it to just two. Beyond reinforcing the truism that attacks only get better, this poses the open problem to improve the attacks presented in this work in terms of login attempt complexity.

References

AMPS22. Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. Four attacks and a proof for Telegram. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 87–106. IEEE, 2022.

BBB+21. Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *2021 IEEE Symposium on Security and Privacy*, pages 777–795. IEEE Computer Society Press, May 2021.

BHP23. Matilda Backendal, Miro Haller, and Kenneth G. Paterson. MEGA: Malleable Encryption Goes Awry. In *44th IEEE Symposium on Security and Privacy, to appear*, 2023.

Ble98. Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 1–12. Springer, Heidelberg, August 1998.

BPH22. Lara Bruseghini, Kenneth G Paterson, and Daniel Huigens. Victory by KO: Attacking OpenPGP using key overwriting. In *ACM Conference on Computer and Communications Security (ACM CCS), to appear*, 2022.

Cop96. Don Coppersmith. Finding a small root of a bivariate integer equation; factoring with high bits known. In Maurer [Mau96], pages 178–189.

[17] In contrast to timing-based side-channel attacks, generally considered less practical over the Internet.

DSvW21. Léo Ducas, Marc Stevens, and Wessel P. J. van Woerden. Advanced lattice sieving on GPUs, with tensor cores. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 249–279. Springer, Heidelberg, October 2021.

How97. Nick Howgrave-Graham. Finding small roots of univariate modular equations revisited. In Michael Darnell, editor, *6th IMA International Conference on Cryptography and Coding*, volume 1355 of *LNCS*, pages 131–142. Springer, Heidelberg, December 1997.

How98. Nicholas A. Howgrave-Graham. *Computational Mathematics Inspired by RSA*. PhD thesis, University of Bath, 1998. Available at `https://researchportal.bath.ac.uk/en/studentTheses/computational-mathematics-inspired-by-rsa`.

KR02. Vlastimil Klima and Tomas Rosa. Attack on private signature keys of the OpenPGP format, PGP(TM) programs and other applications compatible with OpenPGP. Cryptology ePrint Archive, Report 2002/076, 2002. `https://eprint.iacr.org/2002/076`.

LGR21. Julia Len, Paul Grubbs, and Thomas Ristenpart. Partitioning oracle attacks. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 195–212. USENIX Association, August 2021.

LL97. Chae Hoon Lim and Pil Joong Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 249–263. Springer, Heidelberg, August 1997.

Mau96. Ueli M. Maurer, editor. *EUROCRYPT'96*, volume 1070 of *LNCS*. Springer, Heidelberg, May 1996.

May10. Alexander May. Using LLL-reduction for solving RSA and factorization problems. ISC, pages 315–348. Springer, Heidelberg, 2010.

MEG22a. MEGA. About Us. `https://mega.io/about`, September 2022.

MEG22b. MEGA. Mega.nz web client. 2022.

MEG22c. MEGA. Security White Paper. `https://mega.nz/SecurityWhitepaper.pdf`, June 2022.

MEG22d. MEGA. webclient – #15273: Patch for ETH Zurich exploit. `https://github.com/meganz/webclient/commit/d2a0d054d4dbb90f035b3b4b421f780adafaa78e`, June 2022.

MEG22e. MEGA. webclient – #15295: Output detailed information about RSA decoding failures. `https://github.com/meganz/webclient/commit/cd4ab89b2cd0e388b0ea55753b86c8808f810138`, June 2022.

MEG22f. MEGA. webclient – asmcrypto.js. `https://github.com/meganz/webclient/blob/v4.21.4/js/vendor/asmcrypto.js`, August 2022.

MEG22g. MEGA. webclient – asmcrypto.js: Modulus. `https://github.com/meganz/webclient/blob/v4.21.4/js/vendor/asmcrypto.js#L10325`, August 2022.

MEG22h. MEGA. webclient – asmcrypto.js: Modulus_inverse. `https://github.com/meganz/webclient/blob/v4.21.4/js/vendor/asmcrypto.js#L10382`, August 2022.

MEG22i. MEGA. webclient – asmcrypto.js: mredc. `https://github.com/meganz/webclient/blob/v4.21.4/js/vendor/asmcrypto.js#L9706`, August 2022.

MEG22j. MEGA. webclient – asmcrypto.js: RSA_decrypt. `https://github.com/meganz/webclient/blob/v4.21.4/js/vendor/asmcrypto.js#L10746`, August 2022.

MEG22k. MEGA. webclient – crypto.js: api_updfkeysync. `https://github.com/meganz/webclient/blob/v4.21.4/js/crypto.js#L3050`, September 2022.

MEG22l. MEGA. webclient – crypto.js: crypto_decodeprivkey. `https://github.com/meganz/webclient/blob/v4.21.4/js/crypto.js/#L2047`, August 2022.

MEG22m. MEGA. webclient – nodedec.js: crypto_rsadecrypt. `https://github.com/meganz/webclient/blob/v4.21.4/nodedec.js/#L550`, August 2022.

MEG22n. MEGA. webclient – security.js: decryptRsaKeyAndSessionId. `https://github.com/meganz/webclient/blob/v4.21.4/js/security.js#L1231`, August 2022.

MH20. Gabrielle De Micheli and Nadia Heninger. Recovering cryptographic keys from partial information, by example. Cryptology ePrint Archive, Report 2020/1506, 2020. `https://eprint.iacr.org/2020/1506`.

RH22. Keegan Ryan and Nadia Heninger. Cryptanalyzing MEGA in six queries. Cryptology ePrint Archive, Report 2022/914, 2022. `https://eprint.iacr.org/2022/914`.

S^+22. William Stein et al. *Sage Mathematics Software Version 9.5*. The Sage Development Team, 2022. `http://www.sagemath.org`.

SRW22. Alon Shakevsky, Eyal Ronen, and Avishai Wool. Trust dies in darkness: Shedding light on samsung's TrustZone keymaster design. Cryptology ePrint Archive, Report 2022/208, 2022. `https://eprint.iacr.org/2022/208`.

The21. The FPLLL development team. FPLLL, a lattice reduction library. 2021.

The22. The mitmproxy development team. mitmproxy – an interactive HTTPS proxy. 2022.

VR20. Mathy Vanhoef and Eyal Ronen. Dragonblood: Analyzing the dragonfly handshake of WPA3 and EAP-pwd. In *2020 IEEE Symposium on Security and Privacy*, pages 517–533. IEEE Computer Society Press, May 2020.

vW96. Paul C. van Oorschot and Michael J. Wiener. On Diffie-Hellman key agreement with short exponents. In Maurer [Mau96], pages 332–343.

Wik22. Wikipedia. UTF-8. `https://en.wikipedia.org/wiki/UTF-8`, 2022.

A Avoiding aborts in the small subgroup attack

For completeness, here we describe a strategy that allows to choose parameters in the full version of our second attack (Section 4.2) to avoid the abort condition altogether. This strategy only modifies the online part of the attack.

Recall that $R = \prod_{i=0}^{n-1} r_i$. To prevent triggering error $\perp_6$, there are multiple options and here we describe the simplest one. We can set different values of p^*, q^* such that before beginning the attack proper, we can find out which factors are responsible for the error and adapt d' accordingly. In more detail, for each r_i we set $p^* = 2 \cdot r_i \cdot p' + 1$ such that p' is a product of 2-4 large primes, and q^* a large prime. Then, if the client outputs $\perp_6$, it means that with overwhelming probability $d^* \equiv 0 \pmod{r_i}$ for the particular r_i. In this way, we can test all small factors. Once we know which r_i are responsible, we can set a new d' such that $d' \equiv 1 \bmod r_i$ for those r_i and $d' \equiv 0 \bmod r_i$ for the remaining r_i. Unless $\gcd(d^*, p') \neq 1$, which is highly unlikely, this will not produce the error.

Online attack Let $\mathtt{ct}^*, \{g_i\}_{i \in \mathcal{I}}, \{\mathcal{T}_i\}_{i \in \mathcal{I}}$ be the values computed before where $\mathcal{I} = \{0, \ldots, n-1\}$ and let $\mathcal{S} = \emptyset$. Take $\mathtt{q}^*, \mathtt{p}^*, \mathtt{d}', \mathtt{u}^*$ from the precomputation.

1. When the client initiates a login, respond to the client's request with $([\mathtt{k_M}]_{\mathtt{k_e}}, \mathtt{ct}^*, [\mathtt{m}]_{pk}, \mathtt{uh})$, where everything but $\mathtt{ct}^*$ is as it would be in an honest response. If the client does not return $\perp_6$, skip to Step 2. Otherwise, do the following:
   (a) Find a prime q such that $|q|_{\mathtt{b}} = 1024$ and encode it as a byte string $\mathtt{q}$.
   (b) For $i \in \{0, \ldots, n-1\}$, do the following:
      i. Find a prime p_i^* such that $|p_i^*|_{\mathtt{b}} = 1024$ and $p_i^* = 2 \cdot r_i \cdot p' + 1$, where p' is a product of 2-4 large primes. Encode it as a byte string $\mathtt{p}_i^*$.
      ii. Compute $u_i^* \leftarrow q^{-1} \bmod p_i^*$ and encode it as a byte string $\mathtt{u}_i^*$ with $|\mathtt{u}_i^*|_{\mathtt{b}} = 1024$.
      iii. Obtain $\mathtt{ct}_i^* \leftarrow \mathsf{Stitch}(\mathtt{q}, \mathtt{p}_i^*, \mathtt{d}', \mathtt{u}_i^*, [B]_{\mathtt{k_M}})$.
      iv. When the client initiates a login, respond to the client's request with $([\mathtt{k_M}]_{\mathtt{k_e}}, \mathtt{ct}_i^*, [\mathtt{m}]_{pk}, \mathtt{uh})$, where everything but $\mathtt{ct}_i^*$ is as it would be in an honest response.
      v. If the client returns $\perp_6$, add i to $\mathcal{S}$.
   (c) Let $R_{\mathcal{S}} \leftarrow \prod_{i \in \mathcal{S}} r_i$. Compute $d'_{\mathcal{S}} \leftarrow 2^{2047} + 2^{48+128} \cdot \delta + 1$ for $\delta < R$ such that
   $$d'_{\mathcal{S}} \equiv 1 \pmod{R_{\mathcal{S}}}, \text{ and}$$
   $$d'_{\mathcal{S}} \equiv 0 \pmod{R/R_{\mathcal{S}}}.$$
   Encode it as a byte string $\mathtt{d}'_{\mathcal{S}}$ with $|\mathtt{d}'_{\mathcal{S}}|_{\mathtt{b}} = 2048$.
   (d) Replace the precomputed $\mathtt{ct}^*$ with $\mathtt{ct}^* \leftarrow \mathsf{Stitch}(\mathtt{q}^*, \mathtt{p}^*, \mathtt{d}'_{\mathcal{S}}, \mathtt{u}^*, [B]_{\mathtt{k_M}})$.
   (e) For $i \in \mathcal{S}$, we have that $d^* = d' + 2^{48} \cdot B \equiv 0 \pmod{r_i}$ and so $B \equiv 0 \pmod{r_i}$.
2. For $i \in \mathcal{I} \backslash \mathcal{S}$, do the following:
   (a) Follow Step 2a of the full attack in Section 4.2.
   (b) Follow Step 2b of the full attack in Section 4.2.
   (c) We have that $d^* = d'_{\mathcal{S}} + 2^{48} \cdot B \equiv x^{-1} \cdot t_i \pmod{r_i}$, and so $B \equiv \left(2^{48}\right)^{-1} \cdot x^{-1} \cdot t_i \pmod{r_i}$.
3. Then, use CRT to compute $B \bmod R$ from the values collected in Step 1e and Step 2c. This recovers the target plaintext block since $|R|_{\mathtt{b}} \geq 128$.

Cost. If the strategy for handling aborts in Step 1 is executed, it requires up to $12 + 8n$ additional ECB encryption oracle queries as well as n additional logins. However, at the same time the number of overall logins needed can be reduced by $\sum_{i \in \mathcal{S}}(r_i - 1)$ since r_i with $i \in \mathcal{S}$ are excluded from the remainder of the run in Step 2.

B RSA proof of concept implementation

```
# -*- coding: utf-8 -*-
"""
To run:

    sage: bulk_run(2048, 2048//4-16, 1024, seed=0, correct=True,  msb=False, jobs=96, h=36)
    sage: bulk_run(2048, 2048//4-16, 1024, seed=0, correct=False, msb=False, jobs=96, h=36)

"""

from sage.all import (
    ZZ,
    RR,
    PolynomialRing,
    random_prime,
    randint,
    matrix,
    log,
    round,
    ceil,
    set_random_seed,
    inverse_mod,
    gcd,
    cputime,
)
from multiprocessing import Pool
from functools import partial

MAX_REPS = 3


def p_known_msb_instance(bits, unknown_bits=None, correct=True):
    if unknown_bits is None:
        unknown_bits = bits / 6
    while True:
        p = random_prime(2 ** (bits / 2))
        q = random_prime(2 ** (bits / 2))
        if (p * q).nbits() == bits:
            break

    a = p - (p % 2**unknown_bits)

    if not correct:
        p_ = random_prime(2 ** (bits / 2))
        a = p_ - (p_ % 2**unknown_bits)

    return p * q, a


def p_known_lsb_instance(bits, unknown_bits=None, correct=True):
    if unknown_bits is None:
        unknown_bits = bits / 6
    while True:
        p = random_prime(2 ** (bits / 2))
        q = random_prime(2 ** (bits / 2))
        if (p * q).nbits() == bits:
            break

    a = p % 2 ** (bits // 2 - unknown_bits)

    if not correct:
        p_ = random_prime(2 ** (bits / 2))
        a = p_ % 2 ** (bits // 2 - unknown_bits)

    return p * q, a


def p_known_msb_attack_simple(N, a, unknown_bits, bits=None):
    R = 2**unknown_bits
    if bits is None:
        bits = N.nbits() // 2
    A = matrix(ZZ, 3, 3)
    A[0] = [0, R * a, R**2]
    A[1] = [a, R, 0]
    A[2] = [N, 0, 0]
    A = A.LLL()

    P, x = PolynomialRing(ZZ, "x").objgen()

    f = sum(A[0, i] // R**i * x**i for i in range(3))
    r = f.roots()[0][0]

    p_ = gcd((a + r), N)
    return (1 < p_ < N), p_
```

```
def p_known_lsb_attack_simple(N, a, unknown_bits, bits=None):
    if bits is None:
        bits = N.nbits() // 2
    a_ = (inverse_mod(2 ** (bits - unknown_bits), N) * a) % N
    return p_known_msb_attack_simple(N, a_, unknown_bits, bits)


def p_known_msb_attack_full_find_h(bits, unknown_bits):

    Nbits = bits
    xbits = unknown_bits

    gamma = float(xbits) / Nbits

    P, h = PolynomialRing(RR, "h").objgen()
    u = h / 2 - 0.5

    f = gamma * h * (h - 1) - 2 * u * 0.5 * h + u * (u + 1)
    return ceil(f.roots()[1][0])


def p_known_msb_attack_full(
    N, p0, unknown_bits, h=None, u=None, block_size=2, rep=0, bits=None, verbose=False
):
    X = 2**unknown_bits

    if bits is None:
        bits = N.nbits() // 2

    if rep >= MAX_REPS:
        return False, 1

    do_rep = False
    if h is None:
        do_rep = True
        h = p_known_msb_attack_full_find_h(ceil(log(N, 2).n()), ceil(log(X, 2)).n()) + 2 * rep

    if block_size is True:
        block_size = h

    P, x = PolynomialRing(ZZ, "x").objgen()

    if u is None:
        a = 0.5
        u = ZZ(round(a * h - 1 / 2))

    if verbose:
        print(f"h: {h}, u: {u}")

    A = matrix(ZZ, h, h)
    for i in range(h):
        if i < u:
            pi = N ** (u - i) * (p0 + x) ** i
        else:
            pi = x ** (i - u) * (p0 + x) ** u
        for j in range(h):
            A[i, j] = pi[j] * X**j

    A = A.LLL()
    if block_size > 2:
        A = A.BKZ(block_size=block_size, proof=False)

    f = sum(A[0, j] // X**j * x**j for j in range(A.ncols()))
    try:
        r = f.roots()[0][0]
    except IndexError:
        r = 0

    p_ = gcd((p0 + r), N)

    if (p_ == 1 or p_ == N) and do_rep:
        return p_known_msb_attack_full(
            N,
            p0,
            unknown_bits,
            h=None,
            u=None,
            block_size=block_size,
            rep=rep + 1,
            bits=bits,
            verbose=verbose,
        )
    else:
        return (1 < p_ < N), p_


def p_known_lsb_attack_full(N, p0, unknown_bits, bits=None, **kwds):
    if bits is None:
        bits = N.nbits() // 2
    p0_ = (inverse_mod(2 ** (bits - unknown_bits), N) * p0) % N
    return p_known_msb_attack_full(N, p0_, unknown_bits, bits=bits, **kwds)


def testit(seed, bits, unknown_bits, msb=True, correct=True, **kwds):
    set_random_seed(seed)
    if msb:
        N, p0 = p_known_msb_instance(bits, unknown_bits, correct=correct)
    else:
        N, p0 = p_known_lsb_instance(bits, unknown_bits, correct=correct)
    t = cputime()
    if msb:
        res, p = p_known_msb_attack_full(N, p0, unknown_bits, **kwds)
    else:
        res, p = p_known_lsb_attack_full(N, p0, unknown_bits, **kwds)
    print(res, p)
    t = cputime(t)
    return int(res), t


def bulk_run(bits, unknown_bits, repetitions, seed=None, correct=True, msb=True, jobs=1, **kwds):
    if seed is None:
        seed = randint(2**32)

    successes = 0
    total_time = 0.0
    max_time = 0.0

    f = partial(testit, bits=bits, unknown_bits=unknown_bits, msb=msb, correct=correct, **kwds)

    if jobs == 1:
        for i in range(repetitions):
            r, t = f(seed=seed + i)
            successes += r
            total_time += t
            max_time = max(max_time, t)
    else:
        pool = Pool(jobs)
        res = pool.map(f, [seed + i for i in range(repetitions)])
        for r, t in res:
            successes += r
            total_time += t
            max_time = max(max_time, t)
    print(f"rate: {successes / repetitions}, avg t: {total_time / repetitions}, max t: {max_time}")
    return successes / repetitions, total_time / repetitions, max_time
```