Partitioning Oracle Attacks

Julia Len Paul Grubbs Thomas Ristenpart

Cornell Tech

Abstract

In this paper we introduce *partitioning oracles*, a new class of decryption error oracles which, conceptually, take a ciphertext as input and output whether the decryption key belongs to some known subset of keys. Partitioning oracles can arise when encryption schemes are not committing with respect to their keys. We detail adaptive chosen ciphertext attacks that exploit partitioning oracles to efficiently recover passwords and de-anonymize anonymous communications. The attacks utilize efficient key multi-collision algorithms — a cryptanalytic goal that we define — against widely used authenticated encryption with associated data (AEAD) schemes, including AES-GCM, XSalsa20/Poly1305, and ChaCha20/Poly1305.

We build a practical partitioning oracle attack that quickly recovers passwords from Shadowsocks proxy servers. We also survey early implementations of the OPAQUE protocol for password-based key exchange, and show how many could be vulnerable to partitioning oracle attacks due to incorrectly using non-committing AEAD. Our results suggest that the community should standardize and make widely available key-committing AEAD to avoid such vulnerabilities.

1 Introduction

The design of encryption historically separated the goals of confidentiality and authenticity, which led to widespread deployment of encryption schemes vulnerable to chosen-ciphertext attacks (CCAs) [17, 81]. Subsequently, researchers showed how to exploit CCAs to recover plaintext data, most notably via padding [6, 7, 17, 81] and format [12, 26] oracle attacks. As a result, cryptographers now advocate the use of authenticated encryption with associated data (AEAD) schemes and CCA-secure public key encryption. There has since been a shift to adopt fast CCA-secure schemes, notably AES-GCM [58], XSalsa20/Poly1305 [13, 15], and (in the public key setting) hybrid encryption that make use of the aforementioned AEAD schemes.

Such schemes do not target being robust [5, 23], also called committing [29]. While exact formal notions vary, robust/committing schemes ensure that attackers cannot construct a ciphertext that decrypts without error under more than one key. Thus far robustness has not been considered an essential security goal for most cryptographic applications, perhaps because attacks exploiting lack of robustness have only arisen in relatively niche applications like auction protocols [22], or more recently as an integrity issue in moderation for encrypted messaging [21, 29].

We introduce partitioning oracle attacks, a new type of CCA. Briefly, a partitioning oracle arises when an adversary can: (1) efficiently craft ciphertexts that successfully decrypt under a large number of potential keys, and (2) can submit such ciphertexts to a system that reveals whether decryption under a target secret key succeeds. This enables an attacker to learn information about the secret key. The main cryptanalytic step for our attacks is constructing (what we call) key multi-collisions, in which a single AEAD ciphertext can be built such that decryption succeeds under some number k of keys. We formalize this cryptanalytic goal and give an algorithm for computing key multi-collisions for AES-GCM. It builds key multi-collision ciphertexts of length $O(k)$ in $O(k^2)$ time, making them reasonably scalable even to large k. We give more limited attacks against XSalsa20/Poly1305 (and ChaCha20/Poly1305) and AES-GCM-SIV.

Given access to an oracle that reveals whether decryption succeeds, our key multi-collisions for AES-GCM enable a partitioning oracle attack that recovers the secret key in roughly $m + \log k$ queries in situations where possible keys fall in a set of size $d = m \cdot k$. This will not work to recover much information about, e.g., random 128-bit keys where $d = 2^{128}$, but we show that it suffices to be damaging in settings where keys are derived from user-selected passwords or where key anonymity is important.

We explore partitioning oracles via two case studies. First we show how to build a practical partitioning oracle attack against Shadowsocks proxy servers [73]. Shadowsocks was first built to help evade censorship in China, and it underlies other tools such as Jigsaw's Outline VPN [62]. In Shadowsocks, the connections are secured via password-based AEAD with a user-chosen password shared between a client and the proxy server. We show how an attacker can turn the proxy server into a partitioning oracle, despite it being designed to silently drop incorrect ciphertexts.

Simulations using password breach data show that 20% of the time the attacker recovers the user's password by sending 124 ciphertexts to the server — several orders of magnitude fewer than the ∼60,000 required by a standard remote guessing attack. The latter requires less overall bandwidth because our attack ciphertexts are large. However, to succeed 70% of the time, our attack requires fewer queries and less overall bandwidth than the remote guessing attack. We have responsibly disclosed our attacks to the Shadowsocks community, and worked with them to help mitigate the vulnerability.

We then turn to password-authenticated key exchange

(PAKE). Here we focus on incorrect implementations of the OPAQUE [38] protocol, which was recently chosen by the IETF's Crypto Forum Research Group (CFRG) as a candidate for standardization. OPAQUE makes use of an AEAD scheme in its protocol and both the original paper and the (rapidly evolving) standard [46,47] mandate that the AEAD used be committing. We consider what happens when implementations deviate from the standard by using a non-committing AEAD scheme. Indeed, early implementations (some of which predate the standardization effort) use AES-GCM, XSalsa20/Poly1305, or AES-GCM-SIV. As we discuss, these implementations would be hard to use without giving rise to partitioning oracles. Our simulations show that a partitioning oracle here would enable successful password recovery 20% of the time using just 18 man-in-the-middle impersonations against a vulnerable client implementation. Our results therefore reinforce the importance of using committing AEAD by quantifying the danger of failing to do so.

In addition to these in-depth case studies, we discuss other potentially vulnerable cryptographic tools and protocols. Some of these, such as the file encryption tool called age [79] and the internet-draft of the Hybrid Public Key Encryption scheme [10], have already made updates to mitigate our attacks.

Our findings join prior ones [21, 29] in a growing body of evidence that using non-committing AEAD as a default choice can lead to subtle vulnerabilities. We suggest considering a shift towards key-committing AEAD being the default for general use, and using non-committing AEAD only for applications shown to *not* require robustness. This will require some work, however, as existing committing AEAD scheme designs [21,29] are slower than non-committing ones and not yet supported by standards. We believe future work should target fast, committing AEAD schemes suitable for standardization and widespread deployment.

2 Partitioning Oracle Attacks

Here we provide an overview of the abstract partitioning oracle attack setting and example attack scenarios.

Attack abstraction. We consider settings in which an attacker seeks to recover a secret $pw \in \mathcal{D}$ from some set of possible values $\mathcal{D}$. The attacker has access to an interface that takes as input a a bit string V, and uses it plus pw to output the result of some boolean function $f_{pw}\colon \{0,1\}^* \to \{0,1\}$. Here f_{pw} is an abstraction of some cryptographic operations that may succeed or fail depending on pw and V. We use $f_{pw}(V) = 1$ for success and $f_{pw}(V) = 0$ for failure. We give examples of f_{pw} below; in this work f_{pw} usually indicates success or failure of decrypting a ciphertext using password pw.

Given oracle access to adaptively query f_{pw} on chosen values, the question is: *Can an attacker efficiently recover pw*? This of course will depend on f. We refer to f as a *partitioning oracle* if it is computationally tractable for an adversary, given any set $\mathcal{S} \subseteq \mathcal{D}$, to compute a value $\hat{V}$ that partitions $\mathcal{S}$ into two sets $\mathcal{S}^*$ and $\mathcal{S} \setminus \mathcal{S}^*$, with $|\mathcal{S}^*| \leq |\mathcal{S} \setminus \mathcal{S}^*|$, such that $f(pw, \hat{V}) = 1$ for all $pw \in \mathcal{S}$ and $f(pw, \hat{V}) = 0$ for all $pw \in \mathcal{S} \setminus \mathcal{S}^*$. We call such a $\hat{V}$ a splitting value and refer to $k = |\mathcal{S}^*|$ as the degree of a splitting value $\hat{V}$. We say that a splitting value is targeted if the adversary can select the secrets in $\mathcal{S}^*$, in contrast to untargeted attacks that, e.g., compute a splitting value that results in a random partition of S.

For most f_{pw} of practical interest it will be trivial to compute splitting values with degree $k = 1$. In this case, a partitioning oracle attack coincides with a traditional online brute-force guessing strategy for recovering pw. The adversary has nothing other than black-box oracle access to f_{pw} and knowledge of an ordering $pw_1, pw_2, \ldots$ of $\mathcal{D}$ according to decreasing likelihood. First compute a splitting value $\hat{V}_1$ that partitions $\mathcal{S} = \mathcal{D}$ into $\mathcal{S}_1^* = \{pw_1\}$ and the rest of $\mathcal{S}$. Query $f_{pw}(\hat{V}_1)$. The resulting bit indicates whether $\mathcal{S}_1^* = \{pw_1\} = \{pw\}$. Assuming not, compute a splitting value $\hat{V}_2$ that partitions $\mathcal{D} \setminus \mathcal{S}_1^*$ into $\mathcal{S}_2^* = \{pw_2\}$ and the remainder, query $f_{pw}(\hat{V}_2)$, and so on. The attacker will learn pw in worst case $d = |\mathcal{D}|$ oracle queries. Notice that in this case the best possible attack is non-adaptive, meaning the attacker can pre-compute all of its splitting values before it begins.

Partitioning oracles become more interesting when we can efficiently build splitting values of degree $k > 1$. In the limit, we can perform a simple adaptive binary search for pw if we can compute splitting values of degree up to $k = \lceil d/2 \rceil$. Initially set $\mathcal{S} = \mathcal{D}$ and compute a value $\hat{V}_1$ that splits $\mathcal{S}$ into two halves of (essentially) the same size. Query $f_{pw}(\hat{V}_1)$ to learn which half of $\mathcal{D}$ the value pw lies within. Recurse on that half. Like all binary searches, this provides an exponential speed-up over the brute-force strategy because we can recover pw in $\lceil \log d \rceil$ queries. We provide more details about this attack, in particular taking into account non-uniform distributions of the secret pw, in Sections 4 and 5.

Example: Password-based AEAD. Consider a server that accepts messages encrypted using a password pw. To send an encrypted message m, a client derives a key $K \leftarrow \mathsf{PBKDF}(sa, pw)$ using a uniformly random per-message salt sa. It then uses K to encrypt m according to an authenticated encryption with associated data (AEAD) scheme, resulting in a ciphertext C. Here PBKDF is a password-based key derivation function (e.g., one of those specified in PKCS#5 [42]). The client sends $V = (sa, C)$ to the server, which re-derives K and decrypts the ciphertext. This represents a standardized and widely used way to perform password-based AEAD, and it is standard practice now to use fast AEAD schemes such as Galois Counter Mode (GCM) [58] or XSalsa20/Poly1305 [13, 15].

Nevertheless, if the server reveals just whether or not decryption succeeds (e.g., due to an error message), we can construct a partitioning oracle with $f_{pw}(sa, C) = 1$ if and

only if decryption of (sa, C) succeeds. A priori, ciphertext unforgeability would seem to necessarily rule out computational tractability of splitting ciphertexts for degree $k > 1$, but it does not. In fact a simple extension of prior work already gives an attack: Dodis et al. [21] showed how, for any two keys, one can build an AES-GCM ciphertext such that decryption succeeds under both keys. This is possible because AES-GCM is not committing (also called robust [23]). With this, our adversary can check membership in a set $\mathcal{S}_1^* = \{pw', pw''\}$ of two passwords by sending a splitting value $\hat{V}_1$ to the server, as follows. First, it computes keys $K \leftarrow \mathsf{PBKDF}(sa, pw')$ and $K' \leftarrow \mathsf{PBKDF}(sa, pw'')$ for some arbitrary sa. Then, it uses Dodis et al. to construct a ciphertext $\hat{C}_1$ that successfully decrypts under both K and K'. Finally, it sends splitting value $\hat{V}_1 = (sa, \hat{C}_1)$ to the server. If the server's response indicates decryption succeeded, $f_{pw}(sa, \hat{C}_1) = 1$ and $pw \in \mathcal{S}_1^*$. Else, $f_{pw}(sa, \hat{C}_1) = 0$ and $pw \notin \mathcal{S}_1^*$. Iterating this allows finding pw in at most $|\mathcal{D}|/2 + 1$ queries, beating brute-force by almost a factor of two.

We will achieve more significant speed-ups in recovering pw by showing how to build splitting ciphertexts $\hat{C}$ with degree k proportional to $|\hat{C}|$.

Example: password-authenticated key exchange. A classical attack against an early version of the Secure Remote Password (SRP) password-authenticated key exchange (PAKE) protocol [84, 85] can be viewed as a partitioning oracle attack. This attack gives an adversary who engages in the SRP protocol without knowledge of the victim's password the ability to check two password guesses in one run of the protocol. In the parlance of partitioning oracles, the attack turns an SRP client into a partitioning oracle with degree $k = 2$.

We will show in later sections a "k-for-one" (for $k \gg 2$) partitioning oracle attack against incorrect implementations of the OPAQUE PAKE protocol. OPAQUE mandates use of committing AEAD, and the designers clearly specified that using non-committing AEAD leads to vulnerabilities [38]. Nevertheless we found prototype implementations that use AES-GCM and other non-committing AEAD schemes. Our results demonstrate how damaging exploits can be should implementers not abide by the protocol specification.

Example: hybrid encryption. Partitioning oracles can also arise in hybrid encryption. For example, some KEM-DEM constructions, like the HPKE scheme [10] currently being standardized, support authenticating senders based on a pre-shared key (PSK) from a dictionary $\mathcal{D}$ by mixing the PSK into DEM key derivation and using AEAD as the DEM.

If the sender can learn whether the receiver successfully decrypted a ciphertext, a trivial brute-force attack can recover the PSK with enough queries. However, if the DEM is a non-committing AEAD, a malicious sender can gain an exponential speedup by crafting splitting DEM ciphertexts similarly to the password-based AEAD example above. See Appendix A for an example of this attack for HPKE.

Example: anonymity systems. Partitioning oracles against hybrid encryption can also arise in anonymity systems. Prior work showed a link between robustness and anonymous encryption [5, 22, 60]; our partitioning oracle attacks can exploit lack of robustness to perform deanonymization.

As an example scenario consider anonymous end-to-end encrypted messaging, in which a recipient has a key pair (pk, sk) for receiving encrypted messages that are delivered via anonymous channel. A modern choice for encryption would be the `crypto_box` KEM-DEM scheme in the widely-used libsodium [16, 52] library. An adversary wants to determine if the recipient is using one of many possible public keys $\{pk_1, \ldots, pk_d\}$ (possibly gleaned from the web or a public-key directory). The adversary has some way of inferring when an encrypted message is successfully received (e.g., due to a reply message or lack thereof). As above, a brute-force attack over public keys can find the right one in d messages; this may be prohibitive if d is large.

Instead, one can build a partitioning oracle attack against `crypto_box` in this setting requiring only $\log d$ messages. Here $\mathcal{D} = \{1, \ldots, d\}$, that is, the partitioning oracle's secret is which of the keys is used. While we do not know of any deployed system that is vulnerable to this attack scenario, it is possible this vulnerability will arise with growing adoption of non-committing AEAD for E2E encryption.

Discussion. Our results assume that attackers have good estimates of password distributions. Prior work [63] shows that attackers do have good estimates and our experiments follow their simulation methodology. If an attacker wishes to compromise the password of a particular user whose password has never been breached, our attack would fail. However, our simulations show that even with an incomplete password dataset that results in a 20% success rate, hundreds of millions of passwords would be vulnerable.

An interesting aspect of our attack settings is that the attacker has no information about the target secret beyond access to the partitioning oracle and, perhaps, some information about the set $\mathcal{D}$ and how the secret was sampled from it. In particular, our adversaries will not have to break in to some system or observe network communications to obtain a hash or ciphertext derived from pw.

We note that we have framed partitioning oracles as outputting binary values, but it could be possible that there exist oracles that output one of many values. A partitioning oracle that returns one of r values could be used to identify a secret chosen from $\mathcal{D}$ in $\log_r |\mathcal{D}|$ queries. We do not know of any examples of such a partitioning oracle.

Relationship to padding oracles. Partitioning oracle attacks are analogous to, but distinct from, padding oracle attacks [81] or other kinds of format oracle attacks [8, 26]. Partitioning oracles can be exploited to reveal information about secret keys, whereas format oracles can only reveal information about plaintexts. That said, there is some overlap concep-

tually in the underlying techniques, as classic padding oracle attacks like Bleichenbacher's [17] or Vaudenay's [81] can also be viewed as adaptive attacks that provide exponential speed-ups in recovering unknown values.

Additionally, padding oracles may be useful in helping construct partitioning oracles. For example, consider our password-based AEAD example, but replace the AEAD scheme with a scheme such as HMAC-then-Encrypt which is well known to give rise to padding oracle attacks that recover plaintext data [6, 7, 81]. We can use the padding oracle to construct a partitioning oracle where $f_{pw}(\hat{C}) = 1$ if and only if the padding check succeeds. Even if the check succeeds, decrypting $\hat{C}$ will fail, but the padding oracle will reveal f's output and thereby enable recovery of pw.

Relationship to side-channels. Side-channel attacks that exploit timing or other aspects of a computation may help in constructing partitioning oracle attacks. Many padding oracle attacks exploit timing side-channels (e.g., [6]) and they can analogously aid partitioning oracle attacks. One of our attacks against Shadowsocks, for example, exploits a side-effect of correct decryption that is remotely observable. In Section 8, we discuss how timing side-channels that may arise in decryption can enable partitioning oracle attacks, even if a nominally committing scheme is used. But partitioning oracles do not necessarily rely on side channels.

Timing side-channels have also been used recently to learn information about passwords [80] from implementations of the PAKE protocol Dragonfly [31]. We discuss this in more detail in Section 7.

3 Key Multi-Collision Attacks

Our partitioning oracle attacks will utilize the ability to efficiently compute a ciphertext that decrypts under a large number k of keys. We refer to this as a key multi-collision, a cryptanalytic target for encryption schemes that is, to the best of our knowledge, new. Our primary focus will be on key multi-collision attacks against widely used AEAD schemes, including AES-GCM and XSalsa20/Poly1305.

Key multi-collision attacks. We formalize our cryptanalytic goal as follows. Let $\mathsf{AEAD} = (\mathsf{AuthEnc}, \mathsf{AuthDec})$ be an authenticated encryption with associated data scheme, and let its key space be the set $\mathcal{K}$. We write encryption $\mathsf{AuthEnc}_K(N, AD, M)$ to denote running the encryption algorithm with secret key $K \in \mathcal{K}$, nonce N (a bit string), associated data AD (a bit string), and message M (a bit string). Decryption is written analogously, as $\mathsf{AuthDec}_K(N, AD, C)$ where C is a ciphertext. Decryption may output a distinguished error symbol $\perp$. We require of our AEAD scheme that $\mathsf{AuthDec}_K(N, AD, \mathsf{AuthEnc}_K(N, AD, M)) = M$ for all N, AD, M not exceeding the scheme's length restrictions. We formalized AEAD as nonce-based [67], but our treatment and results easily extend to randomized AEAD.

We define targeted multi-key collision resistance (TMKCR) security by the following game. It is parameterized by a scheme AEAD and a target key set $\mathbb{K} \subseteq \mathcal{K}$. A possibly randomized adversary $\mathcal{A}$ is given input a target set $\mathbb{K}$ and must produce nonce N^*, associated data AD^* and ciphertext C^* such that $\mathsf{AuthDec}_K(N^*, AD^*, C^*) \neq \perp$ for all $K \in \mathbb{K}$. We define the advantage via

$$\mathbf{Adv}^{\text{tmk-cr}}_{\mathsf{AEAD},\mathbb{K}}(\mathcal{A}) = \Pr\left[\,\text{TMKCR}^{\mathcal{A}}_{\mathsf{AEAD},\mathbb{K}} \Rightarrow \mathsf{true}\,\right]$$

where "$\text{TMKCR}^{\mathcal{A}}_{\mathsf{AEAD},\mathbb{K}} \Rightarrow \mathsf{true}$" denotes the event that $\mathcal{A}$ succeeds in finding N^*, AD^*, C^* that decrypt under all keys in $\mathbb{K}$. The event is defined over the coins used by $\mathcal{A}$.

We can define a similar untargeted multi-key collision resistance goal, called simply MKCR. The associated security game is the same except that the adversary gets to output a set $\mathbb{K}$ of its choosing in addition to the nonce N^*, associated data AD^*, and ciphertext C^*. The adversary wins if $|\mathbb{K}| \geq \kappa$ for some parameter $\kappa > 1$ and decryption of N^*, AD^*, C^* succeeds for all $K \in \mathbb{K}$. We define the advantage as

$$\mathbf{Adv}^{\text{mk-cr}}_{\mathsf{AEAD},\kappa}(\mathcal{A}) = \Pr\left[\,\text{MKCR}^{\mathcal{A}}_{\mathsf{AEAD},\kappa} \Rightarrow \mathsf{true}\,\right]$$

where "$\text{MKCR}^{\mathcal{A}}_{\mathsf{AEAD},\kappa} \Rightarrow \mathsf{true}$" denotes the event that $\mathcal{A}$ succeeds in finding $\mathbb{K}, N^*, AD^*, C^*$ such that N^*, AD^*, C^* decrypts to non-$\perp$ under all keys in $\mathbb{K}$. The event is defined over the coins used by $\mathcal{A}$.

A TMKCR adversary trivially gives an MKCR adversary, but not vice versa. Both targeted and untargeted MKCR attacks will enable partitioning oracle attacks, as both provide the ability to compute splitting values that work for some subset $\mathbb{K}$ of the key space. But targeted attacks are better for adversaries, since it will allow, for example, generating sets for the most probable keys (e.g., due to a non-uniform distribution over the passwords used to derive them).

Our partitioning oracle attacks will require that decryption fails for $K \notin \mathbb{K}$. This will hold except with tiny probability for the target schemes of interest; thus, we focus on the cryptanalytically hard task of computing the key multi-collisions.

Committing AEAD and MKCR. Informally, a committing encryption scheme is one for which it is computationally intractable to find a pair of keys and a ciphertext that decrypts under both keys. Security goals for committing AE were first formalized by Farshim et al. [23]. Grubbs et al. [29] later formalized committing AEAD, with slightly different semantics than usual for AEAD to capture a goal of compact commitments. Compactness is relevant in the moderation settings they considered, but not here.

The Farshim et al. full robustness (FROB) notion is closest to our MKCR notion: once translated to the nonce-based AEAD setting (by adding nonces and associated data), it is a special case of MKCR in which $|\mathbb{K}| = 2$. We use committing AEAD to refer to schemes that meet this FROB notion, which, in turn, rule out MKCR attacks. The converse is not true, since being MKCR for κ does not imply being MKCR for $\kappa' < \kappa$.

GCM-Enc(K,N,AD,M):

$H \leftarrow E_K(0^{128})$; $P \leftarrow E_K(N \,\|\, 0^{31}1)$
$L \leftarrow \mathsf{encode}_{64}(|AD|) \,\|\, \mathsf{encode}_{64}(|M|)$
$T \leftarrow (L \cdot H) \oplus P$
$m \leftarrow |M|/128$; $a \leftarrow |AD|/128$
$b \leftarrow m + a$
For $i = 1$ to a :
  $T \leftarrow T \oplus (AD[i] \cdot H^{b+2-i})$
For $i = 1$ to m :
  $C[i] \leftarrow E_K(N+1+i) \oplus M[i]$
  $T \leftarrow T \oplus (C[i] \cdot H^{b+2-i-a})$
Return $N \,\|\, C \,\|\, T$

GCM-Dec$(K,AD,N \,\|\, C \,\|\, T)$:

$H \leftarrow E_K(0^{128})$; $P \leftarrow E_K(N \,\|\, 0^{31}1)$
$L \leftarrow \mathsf{encode}_{64}(|AD|) \,\|\, \mathsf{encode}_{64}(|C|)$
$T' \leftarrow (L \cdot H) \oplus P$
$m \leftarrow |C|/128$; $a \leftarrow |AD|/128$
$b \leftarrow m + a$
For $i = 1$ to a :
  $T' \leftarrow T' \oplus (AD[i] \cdot H^{b+2-i})$
For $i = 1$ to m :
  $M[i] \leftarrow E_K(N+1+i) \oplus C[i]$
  $T' \leftarrow T' \oplus (C[i] \cdot H^{b+2-i-a})$
If $T' \neq T$ then return $\perp$
Return M

Multi-Collide-GCM$(\mathbb{K},N,T)$:

$L \leftarrow \mathsf{encode}_{64}(0) \,\|\, \mathsf{encode}_{64}(|\mathbb{K}| \times 128)$
pairs$[\cdot] \leftarrow \perp$; $C \leftarrow \varepsilon$
For $i = 1$ to $|\mathbb{K}|$:
  $H \leftarrow E_{\mathbb{K}[i]}(0^{128})$; $P \leftarrow E_{\mathbb{K}[i]}(N \| 0^{31}1)$
  $y \leftarrow ((L \cdot H) \oplus P \oplus T) \cdot H^{-2}$
  pairs$[i] \leftarrow (H, y)$
$f \leftarrow \mathsf{Interpolate}(\textbf{pairs})$; $\mathbf{x} \leftarrow \mathsf{Coeffs}(f)$
For $i = 1$ to $|\mathbb{K}|$:
  $C \leftarrow C \,\|\, \mathbf{x}[i]$
Return $N \,\|\, C \,\|\, T$

Figure 1: **(Left)** The Galois Counter mode (GCM) encryption and **(middle)** decryption algorithms. **(Right)** The Multi-Collide-GCM algorithm, which takes a set $\mathbb{K}$ of keys, a nonce N, and a tag T and computes a nonce-ciphertext-tag triple $N\|C\|T$ such that it decrypts correctly under every key in $\mathbb{K}$. The function $\mathsf{encode}_{64}(\cdot)$ returns a 64-bit representation of its integer input. The function $\mathsf{Interpolate}(\cdot)$ is a polynomial interpolation algorithm that accepts a vector of data pairs and returns a polynomial, while $\mathsf{Coeffs}(\cdot)$ returns the coefficients of this polynomial. We denote $\cdot$ as multiplication and $\oplus$ as addition in $\mathrm{GF}(2^{128})$.

Related security goals. Multi-collision resistance has been treated in the context of hash functions, but here we are interested in multi-collisions over keys and not over messages. In particular the attacks of Joux [41] are not applicable to our setting, even if one were to focus on keyed Merkle-Damgård hash functions, since applying his attack technique would rely on very long multi-block keys.

One can also formalize and investigate key multi-collision security for other symmetric and asymmetric primitives, including message authentication schemes, digital signatures, and public-key encryption. We leave doing so to future work.

3.1 Key Multi-collisions for AES-GCM

At a high level, our multi-collision attack against AES-GCM reduces the task of finding key multi-collisions to solving a system of linear equations. This is possible because of the algebraic properties of the universal hashing underlying integrity protection in AES-GCM [58, 59].

AES-GCM is an AEAD scheme that composes AES in counter mode with a specially designed Carter-Wegman MAC [82]. The latter uses an XOR-universal hash function called GHASH. Detailed pseudocode is provided in Figure 1. Encryption takes in a nonce N, an AES key K, associated data AD, and plaintext M. It outputs a ciphertext $C_1,\ldots,C_m,T$; here T is the authentication tag and $m = \lceil M/n \rceil$ for $n = 128$ the blocksize of the underlying AES blockcipher denoted by E. The ciphertext blocks $C_1,\ldots,C_m$ are generated using counter mode with E, and the tag T is computed by applying GHASH to AD and $C_1,\ldots,C_m$ to obtain a value h. Finally $T = h \oplus E_K(N \,\|\, 0^{31}1)$. Decryption re-computes the tag, compares it with T, and, if successful, outputs the counter-mode decryption of the ciphertext blocks.

We now explain GHASH, but for simplicity omit associated data. For a key K, GHASH first derives a hash key $H = E_K(0^n)$. It then hashes by computing

$$h = C_1 \cdot H^{m+1} \oplus \cdots \oplus C_{m-1} \cdot H^3 \oplus C_m^* \cdot H^2 \oplus L \cdot H \qquad (1)$$

where C_m^* is C_m concatenated with enough zeros to get an n-bit string and L is an n-bit encoding of the length of the message (equivalently, the length of the ciphertext). The maximum plaintext length is $2^{39} - 256$. The multiplications are performed over the finite field $\mathrm{GF}(2^{128})$ with a particular fixed irreducible polynomial.

Our attack takes as input a set $\mathbb{K} = \{K_1,\ldots,K_k\}$ and nonce N, and produces a single ciphertext $(C_1,\ldots,C_{k-1},T)$ that decrypts correctly under every key in $\mathbb{K}$. For each K_i, we derive the associated GHASH key $H_i = E_{K_i}(0^n)$ and construct a linear equation

$$T = C_1 \cdot H_i^{k-1} \oplus \cdots \oplus C_{k-1} \cdot H_i^2 \oplus L \cdot H_i \oplus E_{K_i}(N \,\|\, 0^{31}1)$$

which one arrives at by assigning H_i to H in (1) and then substituting the result into the equation $T = h \oplus E_{K_i}(N \,\|\, 0^{31}1)$. Note that we have fixed the number of the ciphertext blocks to be $k-1$. The result is then a system of k equations in k unknowns:

$$\begin{bmatrix} 1 & H_1^2 & H_1^3 & \cdots & H_1^{k+1} \\ 1 & H_2^2 & H_2^3 & \cdots & H_2^{k+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & H_k^2 & H_k^3 & \cdots & H_k^{k+1} \end{bmatrix} \cdot \begin{bmatrix} T \\ C_{k-1} \\ \vdots \\ C_1 \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_k \end{bmatrix} \qquad (2)$$

where $B_i = (L \cdot H_i) \oplus E_{K_i}(N \,\|\, 0^{31}1)$. At this point, we can solve the linear equations using Gaussian elimination to produce the desired ciphertext. This will require $O(k^3)$ time, which may be prohibitive for very large k.

The polynomial matrix in (2) is almost a Vandermonde matrix, whose structured form allows for finding solutions

more efficiently. The difference is the missing column $[H_1, H_2, \ldots, H_k]^\intercal$ that is omitted because of the fixed length value L (which we cannot treat as a variable). We can, however, treat T as a fixed value (e.g., a randomly chosen constant) instead of a variable, add one block of ciphertext as a new variable, and solve for the following system of equations

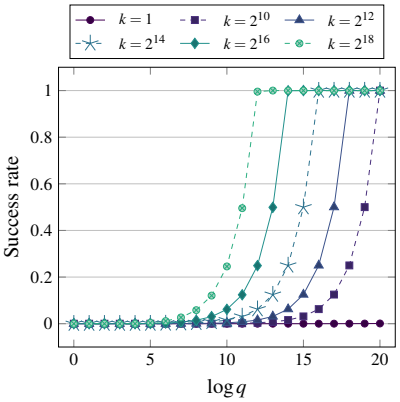
$$\begin{bmatrix} 1 & H_1 & H_1^2 & \cdots & H_1^{k-1} \\ 1 & H_2 & H_2^2 & \cdots & H_2^{k-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & H_k & H_k^2 & \cdots & H_k^{k-1} \end{bmatrix} \cdot \begin{bmatrix} C_k \\ C_{k-1} \\ \vdots \\ C_1 \end{bmatrix} = \begin{bmatrix} B'_1 \\ B'_2 \\ \vdots \\ B'_k \end{bmatrix} \quad (3)$$

where $B'_i = ((L \cdot H_i) \oplus E_{K_i}(N+1) \oplus T) \cdot H_i^{-2}$ and where now L is larger by one block. We can solve this special system of equations in time $O(k^2)$ and space $O(k)$ using off-the-shelf polynomial interpolation algorithms, a factor of k improvement. The resulting solution will have one extra ciphertext block. While ideally an adversary wants multi-collision ciphertexts to be as compact as possible, one extra block will not significantly impact attacks. Detailed pseudocode for this attack is provided in Figure 1. Let $\mathcal{A}_{\text{gcm}}$ be the TMKCR adversary that picks N, T arbitrarily and runs Multi-Collide-GCM.

The adversary is guaranteed to succeed assuming the system of linear equations is solvable, which is equivalent to the matrix having a non-zero determinant. A well-known fact about Vandermonde matrices is that their determinant is non-zero if and only if all the H_i values are pairwise distinct, i.e., $H_i \neq H_j$ for $1 \leq i < j \leq k$. In the ideal cipher model we can therefore directly compute the probability of success (over the coins of the ideal cipher), because in this case the H_i values are chosen uniformly at random, and so $\mathbf{Adv}^{\text{tmk-cr}}_{\text{GCM}}(\mathcal{A}_{\text{gcm}}) \geq 1 - \frac{k^2}{2^n}$. This is essentially one for the values of k we will consider and $n = 128$.

We conjecture that, up to additive constant terms, our attack is "tight" in its trade-off between ciphertext size and runtime: namely, any attack that (w.h.p.) constructs degree-k AES-GCM ciphertexts with fewer than k blocks should require at least birthday-bound complexity. Informally, finding an"unusually short" colliding AES-GCM ciphertext means solving an overdetermined system of equations (i.e. one which has more equations than variables). For such a system to be solvable, there have to be rows that are linear combinations of other rows. Since each column is just increasing powers of a random field element (the hash key $E_K(0^{128})$ for each K in $\mathbb{K}$), this is hard to do assuming the blockcipher acts like an ideal cipher. We leave a formal proof of this to future work.

Performance. We implemented Multi-Collide-GCM using the Python-based mathematics library SageMath [77] and the Magma computational algebra system [18]. We used SageMath for its convenient integration with Python, for which we could utilize cryptography libraries (specifically, PyCryptodome [64]) for AES and for its interface with Magma. We used Magma specifically for its polynomial interpolation algorithm, which we found to be faster than that of SageMath.



k	Time (s)	Size (B)
2	0.18	48
2^{10}	6.6	16,400
2^{12}	29	65,552
2^{14}	170	262,160
2^{16}	1,820	1,048,592
2^{18}	20,122	4,194,320

Figure 2: **(Left)** Success rate of identifying a key uniformly chosen from a set of size $d = 2^{30}$ as a function of the number of queries q for brute-force attack ($k = 1$) and partitioning oracle attack ($k > 1$). **(Right)** Time in seconds to generate key multi-collisions for AES-GCM and the resulting ciphertext size in bytes (including the tag). For $k = 2^{18}$ the time is just for Magma's polynomial interpolation.

Timing experiments were performed on a desktop with an Intel Core i9 processor and 128 GB RAM, running Linux x86-64. We present the results in the table in Figure 2, which shows both the time in seconds to generate a k-way key multi-collision for AES-GCM and the size in bytes of the resulting ciphertext, including the tag. There was little variance in timing when generating multi-collisions, so we report the times for just one execution for each k. Most of the multi-collision ciphertexts could be computed relatively quickly. Colliding ciphertexts for $k = 2^{16}$ keys, for instance, took less than thirty minutes. For smaller k it is much faster. We note that Sage's interface with Magma returns a segmentation fault when polynomial interpolation is used with value $k = 2^{18}$. In Figure 2 for this k value, we therefore report the time to perform polynomial interpolation for 2^{18} randomly-generated points using Magma itself; the timing for the actual attack will be essentially the same.

To illustrate the power of key multi-collisions, we return to the simple PW-based AEAD partitioning oracle scenario described in Section 2. Assume a partitioning oracle that returns $f_K(N, C, T) = 1$ if and only if AES-GCM decryption $\mathsf{AuthDec}_K(N, C \,\|\, T) \neq \perp$. We omit associated data for simplicity. Then, consider an attacker attempting to discover a key chosen uniformly from a set $\mathcal{D}$ of size $d = 2^{30}$ (i.e., the approximate size of a large password dictionary). We simulate the brute-force attack ($k = 1$) assuming the oracle works for plaintexts as small as one byte. We also simulate our adaptive partitioning oracle attack that constructs splitting ciphertexts of size k iteratively for different sets of keys until the oracle returns one. At this point the adversary performs a binary search in $\log k$ queries to find the secret. We perform these simulations for $k \in \{2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}\}$.

The graph in Figure 2 shows the attacks' success rates — how often they succeed in uniquely identifying the key — as a function of the number of queries made. In this context brute-force attacks do poorly, achieving negligible performance even for large numbers of queries. The partitioning oracle attack can search the space much more efficiently, even for moderate k.

We also measured total bandwidth cost (total number of bytes sent to the oracle) used by each attack to achieve a certain success rate. We omitted the nonces from the bandwidth calculations, which can only make the brute-force attack look more competitive with the partitioning oracle attacks. For a 20% success rate, the brute force attack ($k = 1$) has a bandwidth cost of 3.65 GB, while the other values of k require about 3.44 GB. For a 60% success rate, the difference is greater, with the brute force attack accumulating a bandwidth cost of about 11 GB, while the other values of k require only about 10.3 GB.

Ultimately, we conclude that partitioning oracle attacks provide a significant speed up over brute-force search when queries are the limiting factor.

3.2 Other AEAD Schemes

Schemes that use Poly1305. The XSalsa20/Poly1305 [13, 15] and ChaCha20/Poly1305 [14] are widely used AEAD schemes due to their speed, ease of constant-time software implementations, and security properties. Both schemes have a high-level structure similar to AES-GCM, combining a stream cipher (XSalsa20 or ChaCha20) with a Carter-Wegman style MAC called Poly1305. Here we outline a key multi-collision attack against it, and defer the details to the full version of this work.

The core of the attack is against Poly1305 [13], which is similar to GHASH except that it: (1) encodes an input (a ciphertext in the context of its use within the AEAD schemes here) as a sequence of blocks with 0x01 appended; (2) performs the polynomial evaluation over $\mathbb{F}_p$ for prime $p = 2^{130} - 5$ (hence the name); and (3) adds the result to a pseudorandom pad modulo 2^{128} to provide a tag value. The way Poly1305 encodes its inputs breaks the algebraic structure of the collision-finding problem, necessitating a more complex and less scalable attack. Concretely, we were not able to compute splitting ciphertexts with degree greater than ten with our current techniques; this still gives a factor-of-ten speedup in partitioning oracle attacks.

Misuse-resistant AEAD. Many schemes, including those described above, leak information about plaintexts should nonces (IVs) be accidentally reused. Misuse-resistant AEAD [68] provides security even in the presence of nonce reuse. This security goal fundamentally rules out online encryption, meaning one must process the entire plaintext before outputting any ciphertext bits. One popular suggested scheme is AES-GCM-SIV [30], which instantiates the SIV mode of operation [68] using primitives borrowed from AES-GCM (specifically, AES counter mode and a variant of GHASH called POLYVAL).

Nonce misuse-resistance is different than robustness, and in the full version we show that AES-GCM-SIV is vulnerable to key multi-collision attacks. (A variant of this attack, limited to only two keys, was discovered by Schmieg in concurrent work [71].) One interesting point is that our attack against AES-GCM-SIV is not targeted, meaning we cannot precisely control the set of keys that end up in a collision set. As mentioned previously untargeted key multi-collisions suffice for partitioning oracle attacks.

3.3 Passing Plaintext Format Checks

Our MKCR attacks so far ensure that decryption succeeds, but the resulting plaintexts are random. In some cases this suffices, for example when a decryption implementation aborts with an error message when decryption outputs $\perp$. However in some situations — including one of our attacks against Shadowsocks — building partitioning oracles will require MKCR attacks that result in plaintexts that satisfy some format checks.

MKCR with plaintext format checks. We formalize the resulting cryptanalytic goal by extending the MKCR security definition as follows. Let $\mathcal{M}$ be the set of possible plaintexts. We generalize the MKCR game by parameterizing it with a predicate $pr\colon \mathcal{M} \cup \{\perp\} \to \{0,1\}$ that determines whether a message M is valid (i.e., $pr(M) = 1$) or invalid ($pr(M) = 0$). We assume $pr(\perp) = 0$ and that pr is fast to compute.

Then we change the MKCR game to be parameterized by pr, written $\text{MKCR}_{\mathsf{AEAD},\kappa,pr}$. The adversary wins by producing a set $\mathbb{K}$, associated data AD^*, and ciphertext C^* such that $|\mathbb{K}| \geq \kappa$ and for all $K \in \mathbb{K}$ it holds that $pr(\mathsf{AuthDec}_K(AD^*, C^*)) = 1$. This strictly generalizes the prior definition, since we can set $pr(M) = 1$ for all $M \in \mathcal{M}$ and thus arrive at the original same definition. We define the advantage via

$$\mathbf{Adv}^{\text{mk-cr}}_{\mathsf{AEAD},\kappa,pr}(\mathcal{A}) = \Pr\left[\text{MKCR}^{\mathcal{A}}_{\mathsf{AEAD},\kappa,pr} \Rightarrow \mathsf{true}\right]$$

where "$\text{MKCR}^{\mathcal{A}}_{\mathsf{AEAD},\kappa,pr} \Rightarrow \mathsf{true}$" denotes the event that $\mathcal{A}$ wins. The event is defined over the coins used by $\mathcal{A}$.

A rejection sampling approach. Consider a predicate pr and let $p_1 = \Pr[\,pr(M) = 1\,]$ for message M sampled randomly from $\mathcal{M}$. When p_1 is not very small, one simple approach is to use rejection sampling. Consider a target set of keys $\mathbb{K}$. We can choose a random nonce N and tag T and run our MKCR algorithm using $\mathcal{S}, N, T$ to obtain a solution ciphertext $N \,\|\, C \,\|\, T$. We then check that $pr(\mathsf{AuthDec}_K(C,T)) = 1$ for all $K \in \mathcal{S}$. If not, then repeat the attack using a fresh choice of nonce. Each attempt will succeed with probability (negligibly far from) p_1^k for $k = |\mathcal{S}|$, because changing the nonce

leads to fresh pseudorandom plaintexts for each key.

Most format checks will make p_1 too small for this basic approach to work. For example, one of our attacks against Shadowsocks will require the first byte to be a fixed value, making $p_1 = 1/256$. So unless k is small, rejection sampling alone will be too inefficient.

Exploiting structure. We can instead take advantage of the fact that many format predicates will be structured, e.g., checking just the first few bytes of a header. This allows us to extend our AES-GCM attack (and others) in an efficient way. Intuitively we will set aside the ciphertext blocks whose underlying plaintext must satisfy format checks, and leave the rest as free variables to define a system of linear equations.

As a concrete example, assume a predicate pr that only compares the first byte of the plaintext M to some arbitrary fixed byte. We extend our AES-GCM MKCR attack as follows. Consider a potential set of multi-collision keys $\mathcal{S}$. First, choose a nonce N arbitrarily and compute for each $K \in \mathcal{S}$ the first byte of AES-GCM ciphertext. We then determine the largest subset $\mathbb{K} \subseteq \mathcal{S}$ that have the same ciphertext byte value. Applying known results [65] on balls-and-bins problems gives us that $E[|\mathbb{K}|] \approx \frac{|\mathcal{D}|}{256} + 8\sqrt{\frac{|\mathcal{D}|}{256}}$. Then run the targeted TMKCR attack against AES-GCM using N, but fixing the first block of ciphertext to a constant equal to the byte value plus some arbitrary 15 bytes to get a full fixed ciphertext block C_1. Then the system of equations is defined by taking the corresponding contribution to the GHASH equation, namely $C_1 \cdot E_{K_i}(0^{128})^{k+1}$ as a constant and adding it to the right hand side of each equation. One can generalize this to n bits of plaintext, for which $E[|\mathbb{K}|] \approx \frac{|\mathcal{D}|}{2^n} + \sqrt{\frac{2n|\mathcal{D}|}{2^n}}$.

This extension is efficient, running in time in $O(\mathcal{S})$. One could also combine it with the rejection sampling approach by having the first phase try multiple random nonces to look for fortuitous multi-collisions in the first byte, but we did not need to do this for practical attacks.

One can easily extend the approach to other kinds of format checks, though if the check is too constrained it may become inefficient (e.g., if plaintexts must have many fixed bytes). The technique also extends to other stream-cipher based AEAD schemes in a straightforward manner.

4 Password Recovery for Shadowsocks

The prior section showed how to build partitioning oracle attacks against non-committing AEAD schemes. Now we turn to case studies that surface how partitioning oracles arise in practice, enabling password recovery or other harms. We start with Shadowsocks, and show how to build a partitioning oracle that efficiently recovers user-chosen passwords.

Background on Shadowsocks. Originally written by a pseudonymous developer, Shadowsocks [73] is an encrypted proxy for TCP and UDP traffic, based on SOCKS5 [49]. It is used both as a standalone proxy and as the core of other censorship evasion tools such as Google Jigsaw's Outline VPN [62]. The original Github repository has been "starred" by more than 32,000 users and forked by nearly 20,000 [72].

To use Shadowsocks, a user first deploys the Shadowsocks proxy server on a remote machine (typically hosted in a cloud service), provisions it with a static password[1] pw, and chooses an encryption scheme to use for all connections. Originally, only AES-CFB was supported, but cipher choices were modernized after a series of integrity attacks on the protocol [74]. Current documentation recommends either AES-GCM or ChaCha20/Poly1305, which are the only two AEAD schemes supported. Clients given pw can then forward TCP or UDP traffic from their machine to the Shadowsocks proxy. Our attack targets UDP and use of AES-GCM, and so we restrict our explanation to this setup.

The Shadowsocks protocol. The client starts by hashing the user password to obtain a key $K_r = H(pw)$. The hash is currently MD5, but our attacks would work as well should it be replaced with a modern password hashing algorithm. The client then samples a random sixteen-byte salt sa and computes a session key K_s using HKDF [45], as $K_s \leftarrow \text{HKDF}(K_r, sa, \text{"}ss\text{-}subkey\text{"})$. (A new salt and session key are generated for every message.) The client encrypts its plaintext payload pl via $C \leftarrow \mathsf{AuthEnc}(K_s, Z, \varepsilon, \texttt{01} \,\|\, ip \,\|\, port \,\|\, pl)$ where Z denotes a nonce that is set to a string of sufficiently many zero bytes (12 for AES-GCM); the value ε indicates empty associated data; and `01` is a one-byte header indicating that ip is an IPv4 address. The client sends (sa, C) to the server.

When the Shadowsocks server receives (sa, C), it extracts the salt and uses it together with pw to re-derive the session key K_s. It decrypts the remainder of the ciphertext with K_s. If decryption fails, no error message is sent back to the client. Silently dropping invalid or malformed requests is an explicit countermeasure against active probing attacks [83]; it also complicates building partitioning oracles, as we shall see.

If decryption instead succeeds the plaintext's format is checked by verifying that its first byte is equal to `01`.[2] If that check passes, the next six bytes are interpreted as a four-byte IPv4 address ip and two-byte port number $port$. Finally, the rest is sent to the remote server identified by ip and $port$, and the proxy listens on an ephemeral source UDP port assigned by the kernel networking stack for a reply from the remote.

When Shadowsocks receives a reply on the ephemeral port, the server generates a random salt and uses it with pw to generate a new session key. It then encrypts the response, and sends the resulting salt and ciphertext back to the client. The same encryption algorithm is used in both directions.

[1] Using high-entropy symmetric keys instead of passwords became possible recently [75]; this feature does not appear to be widely used.

[2] In fact Shadowsocks supports ASCII domain names and IPv6 addresses, indicated by other byte values, but we ignore these for simplicity.

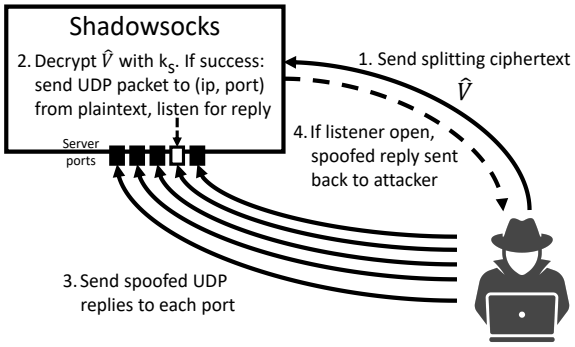


Figure 3: Diagram of the Shadowsocks partitioning oracle. Values $\hat{V}$ and K_s defined in the text. Solid lines indicate actions that always occur, and dashed lines indicate actions that occur only if $\hat{V}$ decrypts correctly, begins with byte `01`, and contains a valid $(ip, port)$ pair.

Threat model. We focus on remote password recovery attacks, meaning a malicious client that knows the IP address of a Shadowsocks server seeks to recover the password(s) it uses. We do not assume the ability to monitor network traffic from honest clients. Capturing honest traffic would enable offline brute-force dictionary attacks against the password-based encryption — future versions of Shadowsocks might consider using password-authenticated public-key encryption instead to mitigate this [19].

A basic attack that works in our threat model is online brute-force, in which the adversary enumerates a sequence of guesses $pw_1, pw_2, \ldots$ and sends an encryption under each guess to the server. By having the encrypted plaintext pl encode a request back to the malicious client, the adversary can determine if decryption succeeds by seeing if it obtains a forwarded request from the proxy. The Shadowsocks designers recommend using rate limits to make remote guessing attacks more difficult, and several of the libraries implement them.

Shadowsocks would be considered secure in our threat model if online brute-force attacks were the best possible attack. We now show how adversaries can do better via partitioning oracles.

Building a partitioning oracle. We now show how to turn a Shadowsocks proxy server into a partitioning oracle. This would be simple if the proxy server responded with an error message when decryption fails, in which case the basic partitioning oracle attack described in Section 2 would apply. But the active probing countermeasure prevents this simple approach. A key insight is that we can exploit the fact that the proxy server opens an ephemeral UDP port in response to a valid request (and does not otherwise). One can view this as a remotely observable, logical side-channel that reveals whether decryption succeeds. See Figure 3 for a diagram of our attack, which we now explain in detail.

The attacker starts with knowledge of a password dictionary $\mathcal{D}$ and an estimate $\hat{p}$ of the probability distribution over passwords in the dictionary. That is, $\hat{p}(pw_i)$ is the probability that $pw_i \in \mathcal{D}$ is the correct password. (We will use password leak data to derive $\hat{p}$, as discussed below.) The attack has two steps, a pre-computation phase and an active querying phase.

Pre-computation phase: In a pre-computation phase, the attacker generates a splitting value (sa^*, C^*), as follows. Given $\mathcal{D}$ with $d = |\mathcal{D}|$ and $\hat{p}$, the attacker uses the MKCR attack that handles format checks from Section 3.3. It derives $K_s^i \leftarrow \mathrm{HKDF}(H(pw_i), sa, \text{“}ss\text{-}subkey\text{”})$ for all $pw_i \in \mathcal{D}$, uses the resulting set $\mathcal{S} = \{K_s^1, \ldots, K_s^d\}$ as the target keys, sets the nonce to be the zero byte string Z, and sets the format check predicate pr to output one if the first byte is equal to `01`. The algorithm outputs a subset of keys $\mathbb{K} \subset \mathcal{S}$ and a ciphertext C^* such that decrypting C^* under each of the keys in $\mathbb{K}$ results in a plaintext with a leading byte equal to `01`.

Applying this directly will not quite work, because Shadowsocks servers will only accept UDP packets whose length is less than or equal to 65,507 bytes. This means we can at best use a key-colliding ciphertext for a key set of size $k = 4,091$. We therefore modify slightly the procedure above to find a size-k subset $\mathbb{K}_{\max} \subset \mathbb{K}$ that has maximum aggregate probability under $\hat{p}$. Fixing a salt sa, we abuse notation and let $\hat{p}(K_s) = \hat{p}(pw)$ for K_s the key derived from pw using sa. Then we solve the optimization problem defined by

$$\mathbb{K}_{\max} = \operatorname*{argmax}_{\mathbb{S} \subseteq \mathbb{K}, |\mathbb{S}| \le k} \sum_{K_s \in \mathbb{S}} \hat{p}(K_s) \; .$$

We compute the key-colliding ciphertext C^* that decrypts under that subset using the first block fixed to ensure the format check is passed. Let $\mathbb{P} \subseteq \mathcal{D}$ be the set of passwords associated to the subset of colliding keys $\mathbb{K}_{\max}$ (for salt sa^*). Recall that since we must fix a block of C^*, C^* will have $k+2$ blocks, including the tag.

Querying phase: Having done the pre-computation, the attacker can then submit to the proxy server (sa^*, C^*) and it will decrypt correctly for any of the 4,091 passwords in $\mathbb{P}$. This is shown as step (1) in Figure 3. Should $pw \in \mathbb{P}$, the server will interpret the decrypted plaintext as a `01` byte followed by a random IPv4 address, destination port, and payload. The IPv4 and destination port will be accepted by the server's network protocol stack with high probability, and so the server will send the payload as a UDP packet to the IP address ip and destination port $port$. It will also open a UDP source port to listen for a response. This is step (2) in the figure.

The attacker does not a priori know the listening port the server uses, and modern operating systems randomize this port. The traditional range used for ephemeral source ports is 49,152 through 65,535, though some systems use slightly larger ranges. The attacker can simply send a UDP probe to every port in that range — the port is left open for five minutes by default for the Shadowsocks server implementations we inspected. This is shown as step (3) in the figure. Should the system respond with ICMP error messages on closed ports, this will already be sufficient for the attacker to learn if a port

was opened. If there is no other activity on the system, then this suffices to construct a partitioning oracle.

But in fact we observed that Shadowsocks server implementations will accept arbitrary response data. Thus, upon receiving the UDP probe the server believes this to be the valid response and proceeds to encrypt it and send it back to the attacker.[3] This is marked as step (4) in the diagram. At this point, the attacker can simply perform trial decryption for each $pw \in \mathbb{P}$ and recover the password.

The attacker can repeat steps (1)–(3) multiple times, focusing iteratively on the set of remaining passwords. The attacker can also amortize the cost of the UDP port scan across multiple attempts, by simply sending a sequence of pre-computed key colliding ciphertexts to the server (for distinct subsets of keys), and then performing the port scan.

Proof of concept. We implemented a proof of concept of the attack. Our experimental setup used a laptop running OS X as a malicious client on a home network, and an EC2 micro instance running Ubuntu 18.04 and go-shadowsocks2 [28]. We used a default configuration of the EC2 instance, except that we allowed UDP inbound traffic on the server's ephemeral port range (32,768–60,999). Without opening those ports, Amazon's firewall will by default block the UDP port scan (because the attacker will not be able to guess the proper source IP and port, which are random).

We verified steps (1)–(4) of the attack work as expected, except that we avoided a port scan (disallowed by Amazon's acceptable use policy without explicit permission) by sending a single UDP packet to the correct port. A real attacker would perform a standard port scan of the ephemeral port range; we confirmed that this works as expected in a local LAN setup (where we had permission to do port scans) using nmap [54]. Computing a key multi-collision ciphertext for $k = 4,091$ took 32 seconds on the same Intel i9 system described in Section 3.1; recall that this is offline pre-computation.

Success rate simulations. To evaluate the efficacy of the attack in recovering a target password, we perform simulations using a sanitized version of a large breach compilation [20] obtained from the authors of [63]. The sanitized dataset contains 1.1 billion passwords together with the frequency with which they occurred. To perform password simulation experiments, we partitioned the password dataset randomly into two halves: a training set ($\mathbb{P}_{train}$) used by the attacker to estimate $\hat{p}$ and a testing set ($\mathbb{P}_{test}$) used as an empirical distribution for sampling a target password pw. This represents an attacker having a good, but not exact, estimate of the distribution from which a password is drawn. The maximum success rate achievable for the simulations is 70%, because the test set has many passwords not found in the training set.

We wrote a program that uses the training set $\mathbb{P}_{train}$ to determine a sequence of password sets $\mathbb{P}_1, \mathbb{P}_2, \ldots$ according

[3] This seems to be a vulnerability in its own right, as it could potentially allow attackers to inject malicious responses to honest client UDP requests.

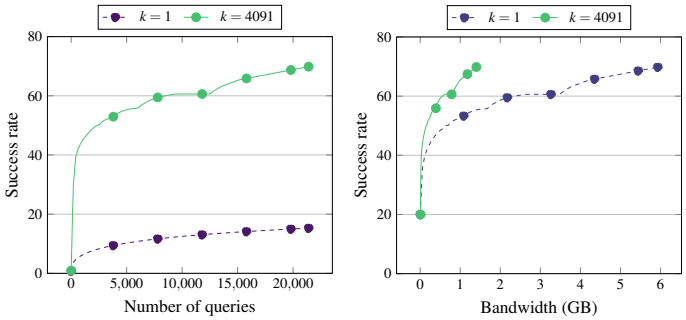


Figure 4: The **(left)** number of queries versus success rate and **(right)** bandwidth versus success rate for simulations of the brute-force attack ($k = 1$) and partitioning oracle attack ($k = 4091$).

to the maximization approach described earlier. Computing the first set (the worst case) took about 704 seconds. The probability of success of this first set is 0.9%. In contrast, the brute-force attack achieves a 0.76% success rate with its first ciphertext. The reason for the mild improvement is that the formatting check for Shadowsocks means that $\mathbb{P}_1$ contains one of the most popular passwords plus many lower probability passwords. One could improve this with further precomputation effort by repeating the process to find higher performing $\mathbb{P}_1$.

Even without such embellishments, the success rate as a function of the number of ciphertext queries made goes up rapidly. The left graph of Figure 4 shows how the partitioning oracle attack outperforms brute force for all query budgets. As examples: the partitioning oracle attack achieves a success rate of 20% with just 124 ciphertexts while brute-force achieves only 3% with the same number. A success rate of 70% would require 21,503 partitioning oracle queries while the brute-force attack would require 87.8 million ciphertexts.

We also estimated bandwidth usage for both attacks, shown in the right graph of Figure 4. A single query in the partitioning oracle attack is 65,532 bytes total, including an 8-byte UDP header, 20-byte IP header, 16-byte salt, and 65,488-byte ciphertext. For the simple brute-force attack a single query is 68 bytes, including the UDP header, IP header, salt, and 24-byte ciphertext. The ciphertext itself includes a 16-byte authentication tag and encrypted 7-byte header and 1-byte payload. For success rates below 25% the brute-force attack requires less total bandwidth than the partitioning oracle attack, but the latter uses less bandwidth above 25%.

Concretely, the total bandwidth of all the submitted ciphertexts in the partitioning oracle attack to achieve 20% success rate would be 8.1 MB across 124 UDP packets. The total bandwidth of submitted ciphertexts to achieve 70% success rate, the maximum possible, would be 1.4 GB across 21,503 UDP packets. The simple online brute-force attack achieves success rate of 20% using 4.1 MB of data sent over 60,250 requests. For 70%, this increases to 5.97 GB of data sent over 87.8 million requests. Note that these calculations do not include the up to 28,231 UDP packets for the port scan of the partitioning oracle attack, but these can potentially be sent

once for multiple (or even all of the) ciphertexts.

To summarize, while partitioning oracle attacks are more expensive computationally, they outperform brute-force in terms of queries and, for larger success rates, bandwidth. This also means that while rate limiting of requests could help mitigate brute-force attacks, it will not be effective against our attack.

Other attack variants. In the full version, we describe a different attack on Shadowsocks servers that support multiple users (with different passwords) on a single port. Because the server identifies the correct key via trial decryption, a "cross-user" key recovery attack is possible.

We were not able to build a working attack that uses TCP connections. The main challenge is that here Shadowsocks servers expect two ciphertexts, first an encryption of the payload length and then an encryption of the payload. The former only allows ciphertexts including 2-byte plaintexts, which is too small for the construction of a splitting ciphertext. As mentioned above deployments use the same password across TCP and UDP, so our UDP attack affects both.

5 Password-Authenticated Key Exchange

We turn now to partitioning oracles in the context of password-authenticated key exchange (PAKE). As noted earlier, a version of the PAKE secure remote password (SRP) protocol [84] has long been known to be vulnerable to a "two-for-one" attack (cf., [85]). An active network adversary impersonates a server response to a client, and based on the client's subsequent behavior can rule out two possible passwords. This provides a modest speedup over brute-force, which rules out one password at a time. We want to know if our techniques can yield bigger speedups in the context of PAKE.

We explore this question in the context of a modern PAKE protocol called OPAQUE [38]. It is undergoing a standardization process currently, having been suggested by the IETF CFRG as a good candidate for next generation PAKE. OPAQUE uses as a component an AEAD scheme, and its designers and the (evolving) draft standards [46, 47] make clear the necessity of using committing AEAD.

We perform a case study focusing on what happens when implementations incorrectly deviate from the specification, and instead use a non-committing AEAD. Indeed some early prototype implementations of OPAQUE use AES-GCM or XSalsa20/Poly1305, as we detail below.

Background on OPAQUE. OPAQUE is meant to replace existing password authentication protocols on the web, which today is done by having the client send the server its password through TLS. This approach requires the server to handle the client's plaintext password, and also relies on public-key infrastructure (PKI) for authentication.

In contrast, OPAQUE is an asymmetric PAKE (aPAKE) that keeps the client's password hidden from the server and does not need PKI to authenticate the server to the client. Asymmetric here means the server only stores the equivalent of a (salted) hash of the password, while the client uses the password directly. OPAQUE provides mutual authentication based on the password. While one can integrate OPAQUE with certs/PKI, we focus on password-only authentication.

OPAQUE works by composing an oblivious PRF (OPRF) [25] with authenticated key exchange (AKE) using a committing AEAD. For space reasons, we defer the reader to [38] for protocol details. Here we follow the OPAQUE description from [38]; recent internet drafts differ in some details that do not affect the attack (should non-committing AEAD be used).

The protocol begins with the server holding an oblivious pseudorandom function (OPRF) key k_s and the user holding password pw. A user registers by sending (over a secure channel) pw to the server. The server computes $rw \leftarrow \mathcal{H}(pw, \mathcal{H}'(pw)^{k_s})$ where $\mathcal{H}'$ hashes strings into a group and $\mathcal{H}$ is any hash function. (This is a standard OPRF construction [39].) The server then chooses a long-term key pair for itself and for the client, uses AEAD with key rw to encrypt the client's key pair and its own public key, and stores its key pair, the client's public key, and the ciphertext C.

After the user has registered, they can initiate a login with the server. The client first chooses an ephemeral public key X_u, computes a blinded OPRF input $\alpha \leftarrow \mathcal{H}'(pw)^r$ for random r, and then sends both values to the server. The server retrieves the client's keys and C, and computes a blinded OPRF output $\beta \leftarrow \alpha^{k_s}$. It chooses its own ephemeral public key X_s and computes the HMQV session key K_{sess}. It sends (β, X_s, C, A_s) to the client, where A_s is a PRF output using K_{sess} (used for session key confirmation). The client can then compute $rw \leftarrow \mathcal{H}(pw, \beta^{1/r})$ and use that to decrypt C to get its long-term key pair. It can then derive the session key K_{sess} as per HMQV and confirm that A_s is correct. The OPAQUE protocol immediately aborts should the client's decryption of C fail.

As discussed in [38], the AEAD must be key-committing because otherwise the client's decryption of C could reveal information about more than one password, similar to the SRP two-for-one attack. Various instantiations of the AEAD have been proposed, including Encrypt-then-HMAC, modifying AES-GCM to add a zeros check, and more.

Early implementations. Despite this guidance, a survey of prototype OPAQUE implementations revealed that a majority use non-committing AEAD. See Figure 5. Many of these prototypes predate the standard drafts, the most recent version of which provides more specific guidance on allowed AEAD schemes. Only one implementation is from a commercial product (opaque-ke [51]); most do not appear to have been reviewed by cryptographers. We therefore expect that future implementations will do better in terms of correctly selecting a committing AEAD. Nevertheless, these indicate that developers need strong, specific guidance about committing

Implementation	AEAD Scheme	MKCR attacks?	Emit errors?
libsphinx [56]	XSalsa20-Poly1305	✓	X
threshold-OPAQUE [61]	XSalsa20-Poly1305	✓	✓
Opaque [53]	XSalsa20-Poly1305	✓	✓
opaque-rs [4]	AES-GCM	✓	X
gustin/opaque [1]	AES-GCM-SIV	✓	✓
gopaque [66]	Encrypt-then-HMAC	X	–
frekui/opaque [48]	Encrypt-then-HMAC	X	–
opaque-ke [51]	AEAD-then-HMAC	X	–
noisat-labs/opaque [2]	NORX	X	–

Figure 5: A summary of early prototype implementations of OPAQUE and the AEAD scheme they use. The righthand column specifies whether the vulnerable implementations emit distinct, explicit error messages during decryption.

AEAD. For instance, Figure 5 shows that XSalsa20-Poly1305, the default authenticated encryption scheme in popular cryptography library `libsodium` [52], is one of the most popular choices for an AEAD scheme. However, it is not committing, and while versions of the OPAQUE documentation explicitly mention that AES-GCM should not be used, no warnings about XSalsa20-Poly1305 have been given. Developers seem unclear about its security properties: one implementation has source code comments stating that a key-committing scheme is necessary right where it uses XSalsa20-Poly1305.

To quantify the danger of such confusion about what AEAD to use, we turn to building partitioning oracles against implementations that use non-committing AEAD.

Building partitioning oracles. We assume the implementation runs the OPRF and AKE in parallel, and that an adversary that can somehow trigger client requests (e.g., via appropriate client-side Javascript [6, 9, 11]), intercept the requests, and respond to them. Upon intercepting a login request, the attacker acts as the OPAQUE server to turn the client into a partitioning oracle f_{pw}. It chooses its own OPRF key k_s^*, and then constructs a splitting value (β, X_s, C^*, A_s). It sets $\beta \leftarrow \alpha^{k_s^*}$, lets A_s be arbitrary, and generates an ephemeral key X_s. Finally it generates a key-multicollision ciphertext C^* for $\mathbb{K} = \{\mathcal{H}(pw, \mathcal{H}'(pw)^{k_s^*})) \mid pw \in \mathcal{S}\}$ for some target set of passwords $\mathcal{S}$. We discuss selecting passwords for $\mathcal{S}$ below. Note that, save β, the splitting value can be pre-computed.

The adversary sends (β, X_s, C^*, A_s) to the client, who will unblind β to obtain a key rw, hash it to derive an AEAD key, and then decrypt C^*. If decryption fails, the client will abort immediately and $f_{pw}(\beta, X_s, C^*, A_s) = 0$; if it succeeds, the client will use the key pair from the plaintext to derive the shared secret k. Then, the client will re-compute A_s' and abort if $A_s' \neq A_s$. If this abort occurs, $f_{pw}(\beta, X_s, C^*, A_s) = 1$.

The difference between the two errors must be visible to the server impersonator to realize the partitioning oracle. We note that the OPAQUE security model [38] and specification allow for distinct error messages (which should be fine when using committing AEAD, but is dangerous here). In Figure 5 the last column marks which vulnerable prototype implementations emit distinct error messages — three of five do. If these messages reach the server impersonator, a partitioning oracle is immediate.

Even without distinct messages, the protocol specifies aborting if decryption fails, then having a separate abort later if the A_s check fails. If implemented with this "early abort", side channels like memory accesses, branch predictors, or timing could reveal which of the two errors occurred.

Measuring the timing channel. To determine whether the potential timing side channel is exploitable, we performed an experiment with libsphinx [56], a more mature prototype that does not emit distinct error messages but does abort early on decryption failure. Most of libsphinx's code is similar to the protocol as described in [38], with two changes that impact timing: (1) it uses a triple-DH handshake instead of HMQV, and (2) it uses the memory- and time-hard Argon2 hash on rw to derive the AEAD key. By default, libsphinx accepts a C^* only up to length 4 MB due to a memory management bug — it crashes for larger ciphertexts due to a statically allocated buffer. Once fixed, it accepts ciphertexts of up to 2 GB. This would enable splitting ciphertexts with degree up to $k = 1.25 \times 10^8$.

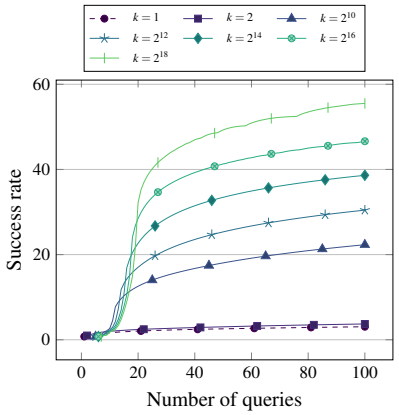
We performed timings for 1000 trials each on a MacBook Pro with a 2.5 GHz Intel Core i7 processor using a static 1 MB key multi-collision ciphertext. The median and mean time were both 121 ms for server responses that did not decrypt properly and 125 ms for server responses that decrypted properly but failed the A_s check. The standard deviation in both cases was 2 ms. This suggests that remote timing attacks should be feasible, though they may require multiple samples per partitioning oracle query to reduce noise (which would reduce attack efficiency by a small factor).

An adaptive partitioning oracle attack. Given the ability to construct a partitioning oracle, the question becomes how to build an attack that extracts the target password pw from the client in as few oracle queries as possible. As for the Shadowsocks attack, consider an attacker that starts with knowledge of a password dictionary $\mathcal{D}$ and an estimate $\hat{p}$ of the password probabilities. Assume k is the maximum multi-collision feasible from our attack, given an implementation's constraint on ciphertext size (e.g., 1.25×10^8 for bug-free libsphinx).

The algorithmic challenge is to develop a search strategy that minimizes the expected number of queries to recover the password. Given input $\mathcal{D}$, q, and k the attacker proceeds as follows. First it finds a subset $\mathbb{P} \subset \mathcal{D}$ that maximally balances the aggregate probability mass of the partition. In other words it solves the following optimization problem:

$$\underset{\mathbb{P} \subset \mathcal{D}, |\mathbb{P}| \leq k}{\operatorname{argmin}} \left| \left(\sum_{pw \in \mathbb{P}} \hat{p}(pw)) \right) - \left(\sum_{pw \in \mathcal{D} \setminus \mathbb{P}} \hat{p}(pw) \right) \right| .$$

This is exactly the optimization version of the partitioning



k	BW	q
1	1.0	60,255
2	1.4	30,085
2^{10}	0.9	69
2^{12}	1.0	27
2^{14}	1.5	18
2^{16}	4.0	18
2^{18}	12.0	19

Figure 6: **(Left)** Success rate achieved for different numbers q of partitioning oracle queries. **(Right)** The maximum total bandwidth (BW) in megabytes and number of queries required to guarantee a 20% success rate.

problem, which is known to be NP-hard but relatively easy to solve (q.v., [44]). Pragmatically for the k, q, and $\hat{p}$ we found that the following simple heuristic works well. First check if the top k passwords by probability have aggregate mass less than 50%. If so, set $\mathbb{P}$ to those top k passwords. Otherwise, perform the classic greedy heuristic that starts with two empty sets $\mathbb{P}, \mathbb{P}'$. Then in order of decreasing probability, add each password to whichever of the two sets has smaller aggregate mass, initially starting with $\mathbb{P}$ and stopping when $|\mathbb{P}| = k$.

The attacker can then use the partitioning oracle with $\mathbb{P}$ as described above to learn if $pw \in \mathbb{P}$. If so it recurses by setting $\mathcal{D} = \mathbb{P}$ and otherwise $\mathcal{D} = \mathcal{D} \setminus \mathbb{P}$.

Attack performance. We use simulations using the datasets described in Section 4 to evaluate the efficacy of the attack, compared to brute force. We compute up to $q = 100$ the set of passwords that will be successfully recovered by the attack for $k \in \{1, 2, 2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}\}$. We then calculate their aggregate probability according to their distribution in $\mathbb{P}_{test}$, yielding the success rate (the percentage of times the attack will succeed). Again note that the maximum success rate is 70% for these simulations.

Figure 6 summarizes the simulation results. The graph (left) shows that in a bruteforce search ($k = 1$), only 3% of passwords can be found with 100 queries. The partitioning oracle attack does significantly better. The curves for $k > 2$ exhibit an initial exponential growth in success rate, which then tapers off to a logarithmic growth. This shift occurs at around $\log_2(k)$ for each value of k because: (1) the first set $\mathbb{P}$ almost always contains the most probable k passwords, and (2) the attack needs around $\log_2(k)$ queries to recover passwords from this set. Growth then tapers off because the popularity of passwords found with further queries decreases.

What this means is that for, e.g., $k = 2^{10}$ which corresponds to a ciphertext length of 16.4 kB, an attacker can achieve 20% success with just 100 queries. For $k = 2^{18}$ the attack obtains 20% with only 19 queries, and 57% with 100 queries.

The right table in Figure 6 shows the total bandwidth and number of queries used by each attack to guarantee a 20% success rate. Despite the linear dependence of k on ciphertext length, partitioning oracles can use about the same bandwidth ($k = 2^{12}$) compared to brute-force search, while decreasing the query cost by $2,228\times$.

Attack viability with TLS integration. We must impersonate the server to build a partitioning oracle. Here we study if the attack still works if OPAQUE is integrated with TLS, as suggested by the paper and a later internet-draft [76].

One suggested integration is to run OPAQUE login within an outer TLS session. The server is authenticated to the client (via TLS's cert auth) before the client begins the OPAQUE login protocol, preventing server impersonation. If the PKI is compromised or circumvented the attack can still work. The draft [76] also suggests using the server's OPAQUE private key for its TLS signature. The server public key is sent to the client in C. (The document notes "there is no need to send a regular TLS certificate".) Because the client must decrypt C before it can check the signature, our attack is possible.

6 Countermeasures

The partitioning oracle attacks against Shadowsocks and non-compliant OPAQUE implementations represent just two examples of a broader problem. We discuss more vulnerable or possibly vulnerable cryptographic tools and protocols in Appendix A, including the Age tool [79], the draft HPKE RFC [10], IKEv1 with passwords as pre-shared secrets [32], password-based encryption in the Java Web Encryption standard [40], and proposed Kerberos extensions [35, 36]. We responsibly disclosed our results to relevant parties, and in several cases worked with developers to explore remediations. Here we discuss these efforts as well as longer-term fixes.

Immediate mitigations. In many cases partitioning oracle vulnerabilities can be mitigated by: (1) length limitations on ciphertexts and/or (2) entropy requirements on shared secrets. For example, in response to our disclosure, the developer of the age tool enforced ciphertext length limits to ensure that splitting ciphertexts generated by our attack can have degree at most $k = 2$ [3]. This limits a partitioning oracle attack to a factor-2 speedup over brute force. The HPKE draft RFC [10], after we disclosed to the authors, was updated to require use of high-entropy secrets, effectively barring human-chosen passwords. This makes the attack infeasible.

When we disclosed our attack to several prominent members of the Shadowsocks community and Outline's tech lead, the Shadowsocks developers took immediate action to disable UDP proxying where it was enabled by default. We discussed possible mitigations at length; because all require a breaking protocol change, the developers elected not to deploy them.

The most recent OPAQUE draft standard specifies an ad hoc committing AEAD scheme, obviating the concern

that future (compliant) implementations will choose a non-committing AEAD scheme. With the current parameter recommendations, the OPAQUE protocol only needs a six-block AE ciphertext; thus, implementions could also limit the ciphertext size as a defense-in-depth measure.

Modifying schemes to be committing. The mitigations above are application-specific, and in some cases they do not completely prevent partitioning oracle attacks. This leaves open the question of how to fix the root cause of vulnerability, the use of non-committing encryption.

One approach would be to attempt to retrofit existing popular AEAD schemes to render them committing. A transform suggested by NIST [78] and an early OPAQUE draft appends an all-zeros block to a message before encrypting with AES-GCM, and, during decryption, checks that resulting plaintext includes the zeros block. This technique can be formally shown to be committing when used with AES-GCM as well as XSalsa20/Poly1305 and ChaCha20/Poly1305. However, security relies on implementations avoiding timing side-channels that allow distinguishing between decryption failure (the authentication tag is wrong) and a zeros-check failure.

Avoiding such timing channels will be difficult given current cryptographic library interfaces. The natural implementation approach is to call a decryption API and only perform the zeros check should that API call succeed. But this may give rise to an observable timing difference, re-enabling the attack: a splitting ciphertext $\hat{C}$ would pass the decryption API and trigger a (failed) zeros check if $f_{pw}(\hat{C}) = 1$ while the zero check would be skipped should $f_{pw}(\hat{C}) = 0$. We performed an experiment to test such a side-channel in the context of a modified OPAQUE implementation. While there was some timing difference, the experiment was ultimately inconclusive. We give more detail in the full version.

Side channels can be avoided if the zeros check happens in decryption before checking the authentication tag. Current APIs for AES-GCM and other schemes cannot partially decrypt a ciphertext (in other contexts this would be dangerous), so libraries will need to be rewritten.

Moving to commiting AEAD. Unfortunately no current standards specify a committing AEAD scheme, such as single-key[4] Encrypt-then-HMAC [29]. We therefore suggest standardizing suitable committing AEAD schemes, including zeros-check variants of AES-GCM and XSalsa20/Poly1305. For general purpose AEAD where the danger of partitioning oracles or other non-committing vulnerabilities (e.g., [21]) cannot be a priori ruled out, we believe committing AEAD should be the default. In particular, all password-based encryption should use committing AEAD.

[4] Using a single key is important: a draft standard [57] for AES-CBC-then-HMAC uses distinct AES and HMAC keys, making it non-committing [29].

7 Related Work

A PAKE protocol by Gentry, MacKenzie, and Ramzan [27] introduced the use of password-based encryption to protect protocol secrets in asymmetric PAKEs. Unlike OPAQUE, which begins with an OPRF, their protocol begins with a symmetric PAKE. The security of the symmetric PAKE rules out a partitioning oracle attack.

Mackenzie [55] gave a PAKE relaxation where a bounded number of guesses can be checked in each impersonation and proved a SPEKE variant [37] allows testing only two passwords per impersonation. This can be viewed as a formal approach for allowing (limited) partitioning oracle attacks.

Two prior attacks on PAKE protocols are relevant to our work. The first is the two-for-one attack [85] on an early version of SRP, mentioned in Section 2. The attack allowed an adversary to check two passwords with one server impersonation. This can be viewed as a partitioning oracle attack, and falls into the more general framework we introduce.

Dragonblood [80] is an attack on the Dragonfly PAKE used in WPA3 [31]. Their attack uses side channels to recover passwords against a WPA3 server, due to a non-constant-time hash-to-curve algorithm that is applied to passwords. They take (remote) measurements and then use that to refine an offline brute force attack against the password, and do not use an adaptive attacks with specially crafted protocol messages to elicit certain behaviors. One could potentially turn the Dragonfly side-channel into a partitioning oracle, which we leave to future work.

Our attacks fall into a broader class of decryption error oracles attacks, which also includes padding oracles attacks [6, 7, 17, 69, 81] and format oracle attacks [8, 26]. All these types of attacks involve adaptive CCAs that enable speeding up recovery of some secret data. Our attacks recover information about decryption keys, rather than plaintexts.

Also related to our work are a series of password-recovery attacks against APOP, an authentication protocol for email, that showed that with server impersonation MD5 collisions can be used to recover a user's APOP password [50, 70]. Their techniques are specific to MD5.

Finally, our multicollision attacks against AES-GCM can be seen as a generalization of the two-key multi-collision used in the invisible salamander attack [21] against Facebook's message franking protocol (q.v., [29]). Our results show how to collide more keys, and identify new places where non-committing encryption leads to subtle vulnerabilities.

8 Conclusion

We introduced partitioning oracle attacks, which exploit a new type of decryption error oracle to learn information about secret keys. We showed how to build AES-GCM ciphertexts that decrypt under a large number of keys, what we

call a key multi-collision attack. We gave more limited attacks against XSalsa20/Poly1305, ChaCha20/Poly1305, and AES-GCM-SIV. In case studies of ShadowSocks and early, non-compliant implementations of the OPAQUE protocol, we demonstrate partitioning oracle attacks that can efficiently recover passwords. We responsibly disclosed the vulnerabilities, and helped practitioners with mitigations.

The non-committing AEAD schemes exploited by our attacks are in wide use, and more tools and protocols are likely to have vulnerabilities. Looking ahead, our results suggest that future work should design, standardize, and add to libraries schemes designed to be committing. A starting point would be to improve the performance of, and work towards standardizing, existing committing AEAD designs [21, 29].

Acknowledgements

The authors thank Hugo Krawczyk for helping us design an early version of the partitioning oracle attack in Section 5 and giving extensive feedback on early drafts of the paper. We also thank Mihir Bellare, Scott Fluhrer, David McGrew, Kenny Paterson, and Chris Wood for helpful feedback on early drafts.

References

[1] opaque. https://github.com/gustin/opaque, 2019.

[2] opaque. https://github.com/noisat-labs/opaque, 2019.

[3] age: mitigate multi-key attacks on ChaCha20Poly1305. https://github.com/FiloSottile/age/commit/2194f6962c8bb3bca8a55f313d5b9302596b593b, 2020.

[4] opaque-rs. https://github.com/Lldenaurois/opaque-rs, 2020.

[5] Michel Abdalla, Mihir Bellare, and Gregory Neven. Robust encryption. In *TCC*, 2010.

[6] Nadhem J Al Fardan and Kenneth G Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE S&P*, 2013.

[7] Martin R Albrecht and Kenneth G Paterson. Lucky microseconds: a timing attack on amazon's s2n implementation of tls. In *EUROCRYPT*, 2016.

[8] Martin R Albrecht, Kenneth G Paterson, and Gaven J Watson. Plaintext recovery attacks against SSH. In *IEEE S&P*, 2009.

[9] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the security of RC4 in TLS. In *USENIX Security*, 2013.

[10] R.L. Barnes, K. Bhargavan, and C. Wood. Hybrid public key encryption, 2020. https://tools.ietf.org/html/draft-irtf-cfrg-hpke-04.

[11] Here come the ⊕ ninjas. https://tlseminar.github.io/docs/beast.pdf, 2011. ekoparty.

[12] Gabrielle Beck, Maximilian Zinkus, and Matthew Green. Automating the development of chosen ciphertext attacks. In *USENIX Security*, 2020.

[13] Daniel J. Bernstein. The Poly1305-AES Message-Authentication Code. In *IACR FSE*, 2005.

[14] Daniel J Bernstein. ChaCha, a variant of Salsa20. In *Workshop Record of SASC*, volume 8, pages 3–5, 2008.

[15] Daniel J. Bernstein. The Salsa20 Family of Stream Ciphers. In *New Stream Cipher Designs - The eSTREAM Finalists*. 2008.

[16] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *LATINCRYPT*, 2012.

[17] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1. In *CRYPTO*, 1998.

[18] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 1997.

[19] Tatiana Bradley, Jan Camenisch, Stanislaw Jarecki, Anja Lehmann, Gregory Neven, and Jiayu Xu. Password-authenticated public-key encryption. In *ACNS*, 2019.

[20] Julio Casal. 1.4 Billion Clear Text Credentials Discovered in a Single Database. 2017.

[21] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In *CRYPTO*, 2018.

[22] Pooya Farshim, Benoît Libert, Kenneth G Paterson, and Elizabeth A Quaglia. Robust encryption, revisited. In *PKC*, 2013.

[23] Pooya Farshim, Claudio Orlandi, and Răzvan Roşie. Security of symmetric primitives under incorrect usage of keys. In *IACR FSE*, 2017.

[24] Dennis Felsch, Martin Grothe, Jörg Schwenk, Adam Czubak, and Marcin Szymanek. The dangers of key reuse: practical attacks on IPsec IKE. In *USENIX Security*, 2018.

[25] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, 2005.

[26] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. Dancing on the lip of the volcano: Chosen ciphertext attacks on Apple iMessage. In *USENIX Security*, 2016.

[27] Craig Gentry, Philip MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In *CRYPTO*, 2006.

[28] go-shadowsocks2. https://github.com/shadowsocks/go-shadowsocks2, 2020.

[29] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In *CRYPTO*, 2017.

[30] Shay Gueron, Adam Langley, and Yehuda Lindell. AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption. *RFC*, 8452, 2019.

[31] Dan Harkins. Dragonfly key exchange (rfc 7664), 2015. https://tools.ietf.org/html/rfc7664.

[32] Dan Harkins, Dave Carrel, et al. The internet key exchange (IKE). Technical report, RFC 2409, november, 1998.

[33] S Hartman and L Zhu. A generalized framework for Kerberos pre-authentication. In *RFC 6113*, 2011.

[34] Heimdal. https://github.com/heimdal/heimdal, 2020.

[35] L. Howard. AEAD encryption types for Kerberos 5. https://tools.ietf.org/html/draft-howard-gssapi-aead-00, 2015.

[36] L. Howard. AEAD modes for Kerberos GSS-API. https://tools.ietf.org/html/draft-howard-gssapi-aead-00, 2015.

[37] David P Jablon. Strong password-only authenticated key exchange. *ACM SIGCOMM Computer Communication Review*, 26(5):5–26, 1996.

[38] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *EUROCRYPT*, 2018.

[39] Stanislaw Jarecki and Xiaomin Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In *TCC*, 2009.

[40] Michael Jones and Joe Hildebrand. JSON web encryption (JWE). *Internet Requests for Comments, RFC*, 7516, 2015.

[41] Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In *CRYPTO*, 2004.

[42] Burt Kaliski. Pkcs5: Password-based cryptography specification version 2.0. Technical report, IETF, 2000.

[43] Charlie Kaufman, Paul Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. Internet key exchange protocol version 2 (IKEv2). Technical report, RFC 5996, September, 2010.

[44] Richard E. Korf. A Complete Anytime Algorithm for Number Partitioning. *Artif. Intell.*, 106(2):181–203, 1998.

[45] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *CRYPTO*, 2010.

[46] Hugo Krawczyk. The OPAQUE asymmetric PAKE protocol. Technical report, Internet-Draft draft-krawczyk-cfrg-opaque-03. Internet Engineering Task Force, 2019.

[47] Hugo Krawczyk. The OPAQUE asymmetric PAKE protocol. Technical report, Internet-Draft draft-krawczyk-cfrg-opaque-05. Internet Engineering Task Force, 2019.

[48] Fredrik Kuivinen. opaque. https://github.com/frekui/opaque, 2018.

[49] Marcus Leech, Matt Ganis, Y Lee, Ron Kuris, David Koblas, and L Jones. RFC1928: Socks protocol version 5, 1996.

[50] Gaëtan Leurent. Message freedom in MD4 and MD5 collisions: Application to APOP. In *FSE*, 2007.

[51] Kevin Lewi and François Garillot. opaque-ke. https://github.com/novifinancial/opaque-ke, 2020.

[52] Libsodium. https://github.com/jedisct1/libsodium, 2020.

[53] George Lyon. Opaque. https://github.com/GeorgeLyon/Opaque, 2019.

[54] Gordon Fyodor Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, 2009.

[55] Philip MacKenzie. On the security of the SPEKE password-authenticated key exchange protocol. IACR eprint, 2001. https://eprint.iacr.org/2001/057.

[56] Stefan Marsiske. libsphinx. https://github.com/stef/libsphinx, 2018.

[57] David McGrew and Kenny Paterson. Authenticated Encryption with AES-CBC and HMAC-SHA. Technical report, Internet-Draft draft-mcgrew-aead-aes-cbc-hmac-sha2-05. Internet Engineering Task Force, 2014.

[58] David McGrew and John Viega. The Galois/Counter mode of operation (GCM). *submission to NIST Modes of Operation Process*, 20, 2004.

[59] David A. McGrew and John Viega. The security and performance of the Galois/Counter Mode (GCM) of Operation. In *INDOCRYPT*, 2004.

[60] Payman Mohassel. A closer look at anonymity and robustness in encryption schemes. In *ASIACRYPT*, 2010.

[61] M. Ember Mou. Opaque. https://github.com/mmou/threshold-OPAQUE/, 2019.

[62] Jigsaw Outline Shadowsocks server. https://getoutline.org/en/home, 2020.

[63] Bijeeta Pal, Tal Daniel, Rahul Chatterjee, and Thomas Ristenpart. Beyond credential stuffing: Password similarity models using neural networks. In *IEEE S& P*, 2019.

[64] PyCryptodome. https://pypi.org/project/pycryptodome/.

[65] Martin Raab and Angelika Steger. "Balls into bins"—a simple and tight analysis. In *RANDOM*, 1998.

[66] Chad Retz. gopaque. https://github.com/cretz/gopaque, 2019.

[67] Phillip Rogaway. Nonce-based symmetric encryption. In *FSE*, 2004.

[68] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In Serge Vaudenay, editor, *EUROCRYPT*, 2006.

[69] Eyal Ronen, Kenneth G Paterson, and Adi Shamir. Pseudo constant time implementations of TLS are only pseudo secure. In *CCS*, 2018.

[70] Yu Sasaki, Lei Wang, Kazuo Ohta, and Noboru Kunihiro. Security of MD5 challenge and response: Extension of APOP password recovery attack. In *CT-RSA*, 2008.

[71] Sophie Schmieg. Invisible salamanders in aes-gcm-siv. https://keymaterial.net/2020/09/07/invisible-salamanders-in-aes-gcm-siv/, 2020.

[72] Shadowsocks server. https://github.com/shadowsocks/shadowsocks, 2020.

[73] Shadowsocks. https://shadowsocks.org/en/index.html, 2020.

[74] SIP004: Support for AEADs implemented by large libraries. https://github.com/shadowsocks/shadowsocks-org/issues/30, 2017.

[75] SIP006: Getting rid of key derivation once and for all. https://github.com/shadowsocks/shadowsocks-org/issues/35, 2017.

[76] Nick Sullivan, Hugo Krawczyk, Owen Friel, and Richard Barnes. Usage of OPAQUE with tls 1.3. Technical report, Internet-Draft draft-sullivan-tls-opaque-00. Internet Engineering Task Force, 2019.

[77] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.0)*, 2020. https://www.sagemath.org.

[78] Meltem Sönmez Turan, Elaine Barker, William Burr, and Lily Chen. Recommendation for password-based key derivation part 1: Storage applications. *NIST Special Publication*, 800(132), 2010.

[79] Filippo Valsorda and Ben Cartwright-Cox. age. https://github.com/FiloSottile/age, 2019.

[80] Mathy Vanhoef and Eyal Ronen. Dragonblood: Analyzing the Dragonfly handshake of WPA3 and EAP-pwd. In *IEEE S&P*, 2020.

[81] Serge Vaudenay. Security flaws induced by CBC padding—applications to SSL, IPSEC, WTLS... In *EUROCRYPT*, 2002.

[82] Mark N. Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.*, 22(3):265–279, 1981.

[83] Philipp Winter and Stefan Lindskog. How the Great Firewall of China is Blocking Tor. In *USENIX FOCI*, 2012.

[84] Thomas D Wu. The secure remote password protocol. In *NDSS*, 1998.

[85] Tim Wu. SRP-6: Improvements and refinements to the secure remote password protocol. Technical report, Submission to the IEEE P1363 Working Group, 2002.

A More (Possible) Partitioning Oracles

We survey several other protocols that may be vulnerable to partitioning oracle attacks. Actual exploitability will depend on implementation and deployment details.

A.1 Password-based and Hybrid Encryption

Kerberos. Two recent internet-drafts suggested the inclusion of AES-GCM and ChaCha20/Poly1305 as available encryption types in Kerberos [35] and GSS-API [36]. They do not appear to have been adopted as RFCs, but the Heimdal library [34] implemented the GSS-API draft. Using these non-committing AE schemes in Kerberos would enable a partitioning oracle attack on Kerberos's encrypted timestamp preauthentication [33], leading to client password recovery. For space reasons, we defer the details to the full version.

Age file encryption tool. Age is a file encryption CLI tool [79] that has a password-based encryption mode. The mode is a KEM-DEM scheme: it uses a password-derived key with ChaCha20/Poly1305 to encapsulate a file key, then computes an HMAC over the KEM (and some metadata) with a key derived from the file key, and then encrypts the plaintext using the file key with ChaCha20/Poly1305. The ciphertext is the KEM and metadata, then the HMAC, then the DEM.

This scheme could be vulnerable to a partitioning oracle attack. Observe that there are three ways for decryption to fail: (1) KEM decryption fails, (2) the HMAC check fails, or (3) DEM decryption fails. If failures (1) and (2) are distinguishable, using a multi-colliding ChaCha20/Poly1305 ciphertext as a KEM could let an attacker check multiple passwords in one decryption. Before we reported this issue, the age implementation did not limit the KEM ciphertext length, thereby allowing key multi-collisions for large key sets.

Javascript Object Signing and Encryption. JOSE is a set of standards for encrypting and authenticating authorization data, such as cookies and access control information. One part of JOSE, the Java Web Encryption (JWE) standard [40], specifies password-based encryption modes that may be vulnerable to an attack similar to the one on age described above. We defer the details to the full version.

Hybrid Public-Key Encryption (HPKE). Recently, the IETF has been evaluating a new standard for hybrid public-key encryption, HPKE [10]. It uses an ECIES-like KEM to derive a DEM key, which is used to encrypt the message. HPKE only supports AES-GCM and ChaCha20/Poly1305 DEMs. It supports a pre-shared secret key (PSK) sender authentication mode by mixing the PSK into the AEAD key derivation. The draft permits short PSKs, but says the scheme is not suitable for use with passwords. If decryption failures are observable to the sender, a partitioning oracle attack can recover the PSK. We defer the details to the full version.

A.2 Authenticated Key Exchange and PSKs

Many widely-used authenticated key exchange (AKE) protocols support PSK authentication. Prominent examples include TLS, the Internet Key Exchange (IKE) used in IPSec, WiFi security protocols like WEP and WPA, WireGuard, and many more. Support for low-entropy PSKs varies between protocols, but none disallows them completely. Next we show that partitioning oracle attacks resulting in PSK recovery could arise on the legacy IKEv1 protocol. Our attack does not extend to more modern AKEs used in IPSec or TLS.

Internet Key Exchange (IKE) v1 PSK. IKEv1 [32] is the first version of the IPSec protocol suite's handshake protocol, and is officially deprecated in favor of version 2 [43], but it is still supported and used for compatibility with legacy devices.

The IKEv1 handshake has three full rounds between the client (called the initiator in IKEv1 parlance) and the server (responder), comprising six messages. After the first two rounds, the client and server have established the shared DH value for the session, but have not yet authenticated each other. Authentication occurs in the fifth and sixth protocol messages; these are the first to be encrypted. The fifth message authenticates the client to the server.

In PSK mode, the client derives the encryption and authentication keys K_e, K_a for the fifth message by computing a PRF, keyed via the PSK, on the shared DH value. Then, it computes the "authentication payload", a hash of the transcript keyed with K_a, encrypts the payload with plain CBC and K_a, and sends the resulting ciphertext to the server. The server re-derives the keys using the shared DH value and the PSK, decrypts the CBC ciphertext, and checks the authentication payload. If this check passes, the server crafts and sends the sixth message to authenticate itself to the client.

Because the server has to decrypt the client's message with a PSK-derived key before authenticating the client, a partitioning oracle attack is theoretically possible. An adversary can initiate an IKEv1 handshake and use the fifth protocol message as a splitting value input to the oracle, and use the server's response as the oracle's output. If the server's responses are different for authentication payload check failure versus other kinds of failures (e.g., packet parsing vulnerabilities) PSK extraction is possible. We have not surveyed IKEv1 implementations or found examples of vulnerable servers; as such, this attack is purely theoretical.

Other AKEs. IKEv1's successor IKEv2 is not vulnerable because of a change to the key schedule. If a PSK was reused or correlated across both IKEv1 and IKEv2, a partitioning oracle on IKEv1 would allow the IKEv2 PSK to be recovered. We do not know of any settings where this happens, but prior work showed that RSA keys were re-used across IKEv1 and IKEv2 in many implementations [24]. We examined the new PSK mode in TLS1.3; it is not vulnerable. For space reasons, we defer an extended discussion to the full version.