Partition Oracles from Weak Key Forgeries

Marcel Armour[1] and Carlos Cid[1,2]

[1] Royal Holloway University of London, Egham, UK
{marcel.armour.2017,carlos.cid}@rhul.ac.uk
[2] Simula UiB, Bergen, Norway

Abstract. In this work, we show how weak key forgeries against polynomial hash based Authenticated Encryption (AE) schemes, such as AES-GCM, can be leveraged to launch partitioning oracle attacks. Partitioning oracle attacks were recently introduced by Len et al. (Usenix'21) as a new class of decryption error oracle which, conceptually, takes a ciphertext as input and outputs whether or not the decryption key belongs to some known subset of keys. Partitioning oracle attacks allow an adversary to query multiple keys simultaneously, leading to practical attacks against low entropy keys (e. g. those derived from passwords).
Weak key forgeries were given a systematic treatment in the work of Procter and Cid (FSE'13), who showed how to construct MAC forgeries that effectively test whether the decryption key is in some (arbitrary) set of target keys. Consequently, it would appear that weak key forgeries naturally lend themselves to constructing partition oracles; we show that this is indeed the case, and discuss some practical applications of such an attack. Our attack applies in settings where AE schemes are used with static session keys, and has the particular advantage that an attacker has full control over the underlying plaintexts, allowing any format checks on underlying plaintexts to be met – including those designed to mitigate against partitioning oracle attacks.
Prior work demonstrated that key commitment is an important security property of AE schemes, in particular settings. Our results suggest that resistance to weak key forgeries should be considered a related design goal. Lastly, our results reinforce the message that weak passwords should never be used to derive encryption keys.

Keywords: Authenticated Encryption · Partitioning Oracles · Weak Key Forgeries · Polynomial Hashing · GCM

1 Introduction

Authenticated Encryption (AE) schemes are designed to provide the core properties of confidentiality and message integrity against chosen-ciphertext attacks (CCA). A particularly important practical class of AE schemes offer Authenticated Encryption with Associated Data (AEAD); AEAD schemes are widely standardised and implemented due to their efficiency and security. As a result of their widespread adoption, AEAD schemes have in some cases been used

in contexts that require additional properties beyond standard CCA security. One particular property that has attracted recent attention is key-commitment [24,18,3], also known as robustness [21], which (informally) states that a ciphertext will only decrypt under the key that was used to encrypt it.

A lack of key-commitment in particular AEAD schemes was exploited by Len et al., who introduced a new class of attack they call "partitioning oracle attacks" [31]. Conceptually, a partitioning oracle takes as input a ciphertext and outputs whether the decryption key belongs to some known subset of keys. Len et al. first construct so-called "splitting ciphertexts" for AES-GCM and ChaCha20Poly1305 that decrypt under every key in a set of target keys. This splitting ciphertext is submitted to a decryption oracle; on observing whether the ciphertext is accepted or rejected, the adversary learns whether or not the decryption key is in the set of target keys. As a result, the adversary is able to query multiple keys simultaneously, speeding up a brute force attack. Combining this with low entropy keys, such as those derived from passwords, results in practical attacks. Len et al. give a number of examples including against Shadowsocks [1], a censorship evasion tool, where the attack results in key recovery.

The concept of weak keys shares some similarities with that of partitioning oracles. Whilst there is no precise definition in the literature, the concept is intuitively clear; Handschuh and Preneel [25] describe a weak key as a key that results in an algorithm behaving in an unexpected way (that can easily be detected) – the idea is that a weak key can be tested for with less effort than brute force. Procter and Cid [36] give a framework that neatly captures weak key forgeries (forgeries that are valid if the key is "weak"), which generalised previous attacks against polynomial hash based message authentication codes (MACs) by Handschuh and Preneel [25] and Saarinen [37]. Procter and Cid's results showed that for these cases the term is a misnomer: in fact, for a polynomial hash based MAC, any set of keys can be considered weak using their forgery techniques.

Abstractly, weak key forgeries and splitting ciphertexts share the same structure: ciphertexts whose successful decryption is contingent on the user's key being in a set of target keys. This suggests that weak key forgeries are a good candidate to carry out partitioning oracle attacks; we show that this is indeed the case. We first generalise the attack formalisation of Len et al. to allow the adversary to act as a machine-in-the-middle, in a more realistic reflection of an attacker's capabilities. As a result we obtain a more abstract definition that encompasses weak key forgeries and splitting ciphertexts. We show how to carry out a partitioning oracle attack using weak key forgeries, and discuss some practical applications of the attack. An advantage of our attack is the control that an adversary obtains over underlying plaintexts, allowing for partitioning oracle attacks in settings that are resistant to the attack of [31], in particular where there are format requirements on underlying plaintexts – including format requirements that are designed to render schemes key-committing. Our results reinforce the conclusions of [31], especially on the danger of deriving encryption keys from user-generated passwords. Furthermore, our results suggest that resistance to weak key forgeries should be considered a related design goal to

key-commitment, particularly in settings that are vulnerable to partitioning oracle attacks. Concretely, our results demonstrate – in contrast to the suggestions of prior work – that adding structure to underlying plaintexts (e.g. packet headers that prefix every plaintext message, or an appended block of all zeros) is not a sufficient mitigation against partitioning oracle attacks.

Related Work. Bellovin and Merritt introduced partition attacks against encrypted key exchange: trial decryption of intercepted traffic allowed multiple keys to be eliminated at once [11]. Other oracle attacks include padding oracles [16,38] or other format oracles [6,4,23]; these attacks are similar to but distinct from partitioning oracles as they recover information regarding plaintexts rather than secret keys. Subverted decryption oracles that reveal information about secret keys were considered in [8,9].

Structure. This paper is structured as follows. After describing our notation below, we provide the relevant background material on polynomial hash based schemes in Section 2. Partitioning Oracle Attacks are introduced in Section 3, and our extension based on Weak Key Forgeries in Section 4. Section 5 describes our experiments with Shadowsocks, as well as other protocols. We close the paper with our conclusions in Section 6.

1.1 Notation

We refer to an element $x \in \{0,1\}^*$ as a string, and denote its length by $|x|$; ε denotes the empty string. The set of strings of length ℓ is denoted $\{0,1\}^\ell$. In addition, we denote by $\perp \notin \{0,1\}^*$ a reserved special symbol. For two strings x, x' we denote by $x \parallel x'$ their concatenation. A block cipher E is a family of permutations on $\{0,1\}^n$, with each permutation indexed by a key $k \in K$, where the key space $K = \{0,1\}^\ell$ for some fixed key length ℓ. The application of a block cipher to input $x \in \{0,1\}^n$ using key k will be denoted by $\mathsf{E}_k(x)$. Arbitrary finite fields are denoted by $\mathbb{F}$, or when we specify its characteristic by $\mathbb{F}_{p^r}$, with p prime.

We use code-based notation for probability and security experiments. We write $\leftarrow$ for the assignment operator (that assigns a right-hand-side value to a left-hand-side variable). If S is a finite set, then $s \leftarrow_\$ S$ denotes choosing s uniformly at random from S. We use superscript notation to indicate when an algorithm (typically an adversary) is given access to specific oracles. If $\mathcal{A}$ is a randomised algorithm, we write $y \leftarrow_\$ \mathcal{A}(x)$ to indicate that it is invoked on input x (and fresh random coins), and the result is assigned to variable y. An experiment terminates with a "stop with x" instruction, where value x is understood as the outcome of the experiment. We write "win" ("lose") as shorthand for "stop with 1" ("stop with 0"). We write "require C", for a Boolean condition C, shorthand for "if not C: lose". (We use require clauses typically to abort a game when the adversary performs some disallowed action, e.g. one that would lead to a trivial win.) We use Iverson brackets $[\cdot]$ to derive bit values from Boolean conditions: For a condition C we have $[C] = 1$ if C holds; otherwise we have $[C] = 0$. In security games we write $\mathcal{A}^{\mathcal{O}_1,\ldots,\mathcal{O}_c} \Rightarrow 1$ to denote the event that the adversary outputs 1 after being given access to the c oracles.

2 Background: Polynomial Hashing

MACs are a symmetric cryptographic primitive that allows two parties sharing a secret key to communicate with the assurance that their messages have not been tampered with. Many popular MAC schemes are constructed from universal hash functions that are realised by polynomial evaluation; such MACs based on polynomial hashing are discussed in Section 2.1. They are often used to provide the authentication component for AEAD schemes, which are discussed in Section 2.2, where we give an overview of the two most widely used polynomial hash based AEAD constructions, McGrew and Viega's Galois/Counter Mode (GCM) [32] and Bernstein's ChaCha20-Poly1305 [12].

2.1 MACs from Polynomial Hashing

A polynomial hash based authentication scheme is built on a family of universal hash functions that are based on polynomial evaluation. It takes as input an authentication key H and message M (consisting of plaintext or ciphertext blocks depending on context). Let $M = M_1 \parallel \cdots \parallel M_p \parallel M_{p+1}$ with $M_{p+1} = \mathrm{len}(M)$ and all M_i considered as elements of a field $\mathbb{F}$ (typically $\mathbb{F}_{2^n}$), and $g_M(x)$ be the polynomial in $\mathbb{F}[x]$ defined as $g_M(x) = \sum_{i=1}^{p+1} M_i x^{p+2-i}$. If we also consider $H \in \mathbb{F}$, the polynomial hash $h_H(M)$ of M is calculated by evaluating $g_M(x)$ at H, i.e.

$$h_H(M) := g_M(H) = \sum_{i=1}^{p+1} M_i H^{p+2-i} \in \mathbb{F}\,.$$

The hash value is usually encrypted with a pseudo-random one-time pad, to provide the output authentication tag.

The underlying properties of polynomials are inherited by the hash function and thus the authentication scheme; in particular, the fact that adding a zero valued polynomial will not change the value of the hash (which gives rise to "weak key" forgeries, discussed in Section 2.4) and the fact that it is possible to construct a polynomial that passes through a set of given points (giving rise to multi-key collisions, discussed in Section 2.3).

2.2 AEAD

Let $\mathsf{AEAD} = (\mathsf{AuthEnc}, \mathsf{AuthDec})$ be an AEAD scheme, and let its key space be the set K. Encryption takes as input a key $k \in K$, together with a tuple of nonce, associated data and plaintext (N, D, P) and returns a ciphertext and authentication tag. We write $(C, T) \leftarrow \mathsf{AuthEnc}_k(N, D, P)$. Similarly, decryption takes as input a key $k \in K$ together with a tuple (N, D, C, T) of nonce, associated data, ciphertext and authentication tag and returns either a message or the error message $\perp$ to indicate that the decryption was not successful. Correctness requires that for all N, D, P not exceeding the scheme's length restrictions, $\mathsf{AuthDec}_k(N, D, C, T) = P$ with $(C, T) = \mathsf{AuthEnc}_k(N, D, P)$.

A common paradigm for constructing AEAD schemes is to use an Encrypt-then-MAC (EtM) construction with a stream cipher for encryption and an authentication component from a polynomial based universal hash function. We give a brief overview of the most widely adopted and standardised schemes: McGrew and Viega's AES Galois/Counter Mode (AES-GCM) [32] and Bernstein's ChaCha20-Poly1305 [12,35].

AES-GCM. AES-GCM encryption takes as input: an AES key k, a nonce N, plaintext $P = P_1 \parallel \cdots \parallel P_p$ and associated data $D = D_1 \parallel \cdots \parallel D_d$. The key is 128, 192 or 256 bits long, the nonce N should preferably be 96 bits long although any length is supported. For each i, $|P_i| = |D_i| = 128$ except for perhaps a partial final block. With this input, AES-GCM returns a ciphertext $C = C_1 \parallel \cdots \parallel C_p$ (the same length as the plaintext) and an authentication tag T. From here on, we will omit associated data for simplicity. The plaintext is encrypted using an instance of the AES in counter mode, under key k with counter value starting at CTR_1. If the nonce is 96 bits long the initial counter value (CTR_0) is $N \parallel 0^{31}1$, otherwise it is a polynomial evaluation-based hash of N after zero padding (using the hash key described below). For each i, $\mathrm{CTR}_i = \mathrm{inc}(\mathrm{CTR}_{i-1})$, where $\mathrm{inc}(\cdot)$ increments the last 32 bits of its argument (modulo 2^{32}).

The authentication tag is computed from GHASH, a polynomial evaluation hash (in $\mathbb{F}_{2^{128}}$). The ciphertext C is parsed as 128-bit blocks (with partial final blocks zero padded) and each block is interpreted as an element of $\mathbb{F}_{2^{128}}$. We denote by L an encoding of the length of the (unpadded) ciphertext and additional data. The hash key H is derived from the AES block cipher key: $H = \mathsf{E}_k(0^{128})$. The hash function is then computed as:

$$h_H(C) = L \cdot H \oplus C_p^* \cdot H^2 \oplus C_{p-1} \cdot H^3 \oplus \cdots \oplus C_2 \cdot H^p \oplus C_1 \cdot H^{p+1}, \quad (1)$$

where all operations are in $\mathbb{F}_{2^{128}}$, and C_p^* denotes the zero-padded last block. The authentication tag is given by: $T = \mathsf{E}_k(\mathrm{CTR}_0) \oplus h_H(C)$.

ChaCha20-Poly1305. Poly1305 is similar to GHASH, and to form AEAD schemes it is most commonly combined with the ChaCha20 stream cipher [13], although Poly1305-AES is also an option [12]. For concreteness, we will give a description of ChaCha20-Poly1305 and note that the differences are trivial.

ChaCha20-Poly1305 encryption takes as input: a 32-byte ChaCha20 key k, a 12-byte nonce N, plaintext P and additional data D. With this input, ChaCha20-Poly1305 returns a ciphertext C (the same length as the plaintext) and an authentication tag T of length 16 bytes. From here on, we will omit the associated data for simplicity. First, the plaintext is divided into 64 byte blocks, except perhaps for a partial final block, and encrypted using the ChaCha20 stream cipher, under key k.

The authentication tag is next computed from a polynomial evaluation hash in the finite field $\mathbb{F}_{2^{130}-5}$. The ciphertext to be hashed is divided into 16-byte blocks with any partial final block zero-padded to 16 bytes. We denote by L an encoding of the (unpadded) ciphertext and additional data. Each block is encoded as an integer modulo $2^{130} - 5$ by first appending `0x01` to each block, and interpreting the resulting block as a little-endian integer X_i.

The authentication tag is computed from a polynomial evaluation hash (in $\mathbb{F}_{2^{130}-5}$). First we derive the hashing key r and the pseudo-random one time pad s: the first 32 bytes of $H = \mathsf{E}_k(N_0 \parallel N)$ is divided into two 16-byte strings $\tilde{r}$ and s. Here N_0 represents 0 encoded as a 4-btye little-endian integer.

The hashing key r is obtained from the string $\tilde{r}$ by setting some of the bits to zero in a process referred to as "clamping"; we gloss over the specific details. The hash function is then computed as

$$h_r(C) = L \cdot r \oplus C_p^* \cdot r^2 \oplus C_{p-1} \cdot r^3 \oplus \cdots \oplus C_2 \cdot r^p \oplus C_1 \cdot r^{p+1},$$

where all operations are in $\mathbb{F}_{2^{130}-5}$, and C_p^* denotes the last zero-padded block. The authentication tag is given by:

$$T = (s \oplus h_r(C)) \mod 2^{128},$$

where s and $h_r(C)$ are interpreted as elements of $\mathbb{F}_{2^{128}}$, and the result as an integer modulo 2^{128}.

2.3 Key Commitment

A committing AE scheme is one which satisfies the property of *key commitment*, which (informally) states that a ciphertext will only decrypt under the key that was used to encrypt it. Equivalently, for a committing AE scheme, it should be infeasible to find a ciphertext that will decrypt under two different keys. Security goals for committing AE were first formalised by Farshim et al. [21] under the name "robustness". Although key commitment is not part of the design goal of AE schemes, there are natural scenarios where a lack of key commitment results in security issues. Dodis et al. [18] and Grubbs et al. [24] show how to exploit non-committing AE schemes in the context of abuse reporting in Facebook Messenger. Albertini et al. [3] give some further practical examples where a lack of key commitment leads to practical attacks, e. g. in the setting of paywalled subscription material where a malicious publisher might prepare a ciphertext that decrypts to different content for different users.

The partitioning oracle attack of Len et al. [31] exploits the inherent lack of key commitment for polynomial hash based AEAD schemes. They construct a ciphertext $\hat{C}$ that decrypts under every key in a set of target keys $\mathbb{K}^* = \{k_1, \cdots, k_\ell\}$ by constructing a linear equation whose variables are the blocks of ciphertext; $\hat{C}$ is the solution to the equation. We describe the technique using AES-GCM for concreteness.

Given $\mathbb{K}^*$ and nonce N, first derive the associated GHASH key $H_i = \mathsf{E}_{k_i}(0^n)$ for each $k_i \in \mathbb{K}^*$. Then construct the linear equation

$$T = C_1 \cdot H_i^{p-1} \oplus \cdots \oplus C_{p-1} \cdot H_i^2 \oplus L \cdot H_i \oplus \mathsf{E}_{k_i}(N \parallel 0^{31}1),$$

which is arrived at by assigning H_i to H in eq. (1) and substituting the result into the expression for the tag $T = h_H(C) \oplus \mathsf{E}_{k_i}(\mathrm{CTR}_0)$. The result is a system of ℓ equations in ℓ unknowns which can be solved; this can be done more efficiently

using a clever trick (fixing T and adding one block of ciphertext as a new variable, giving a Vandermonde matrix). We refer the reader to [31] for further detail.

Generic AE solutions, the so-called *generic composition* constructions such as Encrypt-then-MAC, can provide key-commitment, as shown by Farshim et al. [21] who suggested using a keyed hash function such as HMAC [10] for authentication. However, if a key-committing scheme is required for security in some particular setting, then performance considerations may mean that switching to e. g. encrypt-then-HMAC is not practical. This is illustrated by the choice of Facebook Messenger to use AES-GCM to encrypt message attachments despite work showing that this was insecure. Albertini et al. [3] propose two generic fixes that minimise the changes needed to add key-commitment to widely deployed, highly efficient schemes such as AES-GCM:

1. **Padding Fix**. Prepend a constant string to messages before encrypting; check for the presence of the constant string after decrypting. This fix is also given in an early draft of an OPAQUE protocol RFC [30], and discussed in [31]. This solution – essentially adding redundancy to the message – is not generically secure and must be analysed per scheme. Albertini et al. [3] perform this analysis for AES-GCM and ChaCha20-Poly1305, showing that in both cases the resulting scheme is key-committing.
2. **Generic Fix**. From a given key k, derive an encryption key $k_{\text{enc}} = F_{\text{enc}}(k)$ and a commitment to the key $k_{\text{com}} = F_{\text{com}}(k)$. Here F_{enc} and F_{com} are collision resistant hash functions. Ciphertexts for the resulting key-committing scheme consist of a regular ciphertext (for the underlying AEAD scheme) together with the commitment to the key. Albertini et al. [3] show that this construction provides key-commitment, if the functions F_{enc} and F_{com} used to derive the encryption key and commitment are collision resistant pseudorandom functions.

2.4 Weak Key Forgeries

In symmetric cryptography, a class of keys is called a *weak key class* if the algorithm behaves in an unexpected way when operating under members of that class, and this behaviour is easy to detect. In addition, identifying that a key belongs to such a weak key class should require trying fewer than N keys by exhaustive search (or verification queries), where N is the size of the class [25]. In the context of polynomial hash based authentication schemes, e.g. the GCM mode, Handschuh and Preneel [25] and Saarinen [37] identified several weak key classes. In [36], Procter and Cid proposed a generic framework to mount forgery attacks against polynomial-based MAC schemes based on weak keys. Their framework encompasses the previous forgery attacks from [25] and [37], as well as the earlier Joux's Forbidden Attack [27], and is based on a malleability property present in polynomial-based MAC schemes.

If h_H is a polynomial hash under key H and M is a message input, let $h_H(M) = g_M(H)$, where $g_M(x) = \sum_{i=1}^{p+1} M_i x^{p+2-i} \in \mathbb{F}[x]$ and $H \in \mathbb{F}$ (as

in Section 2.1). Now let $q(x) = \sum_{i=1}^{p+1} q_i x^{p+2-i} \in \mathbb{F}[x]$ be a polynomial with constant term zero, such that $q(H) = 0$. Then

$$h_H(M) = g_M(H) = g_M(H) + q(H) = g_{M+Q}(H) = h_H(M + Q),$$

where $Q = q_1 \parallel q_2 \parallel \ldots \parallel q_\ell$ and the addition $M + Q$ is done block-wise[3]. It follows that given a polynomial $q(x)$ satisfying these properties, it is straightforward to construct collisions for the hash function. In fact, we have that $q(x)$ is in the ideal $\langle x^2 - Hx \rangle$, and any polynomial in this ideal can be used to produce collisions. On the other hand, collisions in the hash function correspond to MAC forgeries, by substituting the original message for the one that yields a collision in the polynomial hash. Thus this method allows an adversary to create forgeries when they have seen a tuple of (nonce, message, tag), by simply modifying the message, as above. Saarinen's cycling attacks [37] are a special case of this attack. Forgeries for GCM and variants are presented in [36]. Later, an efficient method for constructing forgery polynomials which have disjoint sets of roots (i.e. keys) was proposed in [2].

3 Partitioning Oracle Attacks

Partitioning oracles, introduced by Len et al. [31] are a class of decryption error oracles which, conceptually, take a ciphertext and return whether the decryption key belongs to some known subset of keys. This allows an adversary to speed up an exhaustive search by querying multiple keys at once; in effect, partitioning the key space. The approach of [31] relies on two conditions: (1) the non-key committing property of polynomial hash based AE schemes is exploited to craft targeted "splitting" ciphertexts that will decrypt under multiple keys; and (2) a decryption oracle that reveals whether decryption (with the user's key) of such a splitting ciphertext succeeds or not.

Abstractly, a partitioning oracle will (in the optimal case) allow a binary search of the key space, giving a logarithmic improvement over naïve exhaustive search. This requires being able to query half the keys in the key space. In practice however, there is a limit to the number of keys that can be queried at once – e. g. for AES-GCM, messages are required to be less than approx. 64GB ($2^{39} - 256$ bits [20]), and applications may impose further restrictions depending on context. Nevertheless, as shown in [31], it is still possible to launch practical attacks by combining partitioning oracles with knowledge of non-uniform key distributions, which arise in particular when human memorable passwords are used to derive keys, and can be estimated from password breaches [33].

We note that the conditions for a partitioning oracle attack can be satisfied with weak key forgeries, following the work of Procter and Cid [36] (see Section 2.4). Weak key forgeries require a valid ciphertext to construct the forgery; a crucial difference to [31], which considers adversaries that only have access to a decryption oracle. In practice this is a limitation of the adversary that does not

[3] The shorter message is zero-padded if required.

tally with observed adversarial strategies against censorship evasion [14,39]. We thus extend the model by allowing an adversary to obtain valid ciphertexts from chosen plaintexts, a standard adversarial model for AE. In fact, this assumption is stronger than required; as we later show, adversaries with only "machine-in-the-middle" capabilities can carry out effective partitioning oracle attacks using weak key forgeries. Known and chosen plaintext capabilities lead to more powerful attacks, as we briefly describe in Section 5.1.

Example: generic encryption. Consider a client and server communicating with end-to-end encryption, using an AEAD scheme and a shared key k derived from password pw. The client encrypts message P (together with any associated data D), using key k and nonce N to obtain a ciphertext tag pair $(C, T) \leftarrow \mathsf{AuthEnc}_k(N, D, P)$. The conditions for a partitioning oracle attack are met if the server reveals whether or not decryption succeeds; it might for example output an observable error message, or reveal the information via a side-channel.

Example: Password-authenticated Key Exchange. A Password Authenticated Key Exchange (PAKE) is a cryptographic key exchange protocol in which a client authenticates to a server using a password pw that the server has stored (as the equivalent of a hash). Len et al. show how to launch a partitioning oracle attack against OPAQUE, a modern PAKE protocol currently undergoing standardisation. OPAQUE uses an AEAD scheme as a component, and Len et al. show the necessity of the AEAD scheme being key-committing by considering deviations from the specification in some early prototype implementations. OPAQUE works by composing an oblivious PRF with an authenticated key exchange; Len et. al.'s attack relies on the fact that the server sends a ciphertext C encrypted using the password during an execution of the protocol.

3.1 Attack Abstraction: Formal Definition of a Partitioning Oracle

Following [31], we consider settings in which an attacker targets AE and seeks to recover a user's key $k \in K$, where the key is deterministically derived from secret password $pw \in \mathcal{D}$. We write $K(\mathcal{D}) \subseteq K$ for the set of keys derived from passwords and $k(pw) \in K(\mathcal{D})$ to denote a key derived from password pw. The attacker is given access to an interface that takes as input ciphertext C, and outputs whether or not the ciphertext decrypts correctly (passing any format checks) under the user's key $k(pw)$. The attacker is further given access to an interface that will encrypt plaintexts of the attacker's choosing and return the ciphertext. This set-up represents a "partitioning oracle" if it is computationally tractable for the adversary, given any set $\mathbb{K} \subseteq K(\mathcal{D})$, to compute a value $\hat{C}$ that partitions $\mathbb{K}$ into two sets $\mathbb{K}^*$ and $\mathbb{K} \setminus \mathbb{K}^*$, with $|\mathbb{K}^*| \leq |\mathbb{K} \setminus \mathbb{K}^*|$, such that $\mathsf{AuthDec}_k(\hat{C}) \neq \perp$ for all $k \in \mathbb{K}^*$ and $\mathsf{AuthDec}_k(\hat{C}) = \perp$ for all $k \in \mathbb{K} \setminus \mathbb{K}^*$. We call such a $\hat{C}$ a *splitting ciphertext* and refer to $|\mathbb{K}^*|$ as the degree of $\hat{C}$. We distinguish between targeted splitting ciphertexts, where the adversary can select the secrets in $\mathbb{K}^*$, and untargeted attacks.

In general, the definition can be applied to arbitrary cryptographic functionalities by considering a Boolean function f that takes as input a string and a

key, returning 1 if some cryptographic operation succeeds and 0 otherwise. The attacker has access to an interface that takes as input a bit string V, and uses it plus k to output the result of some Boolean function $f_k : \{0,1\}^* \to \{0,1\}$. Here f_k is an abstraction of some cryptographic operations that may succeed or fail depending on k and V; set $f_k(V) = 1$ for success and $f_k(V) = 0$ for failure. We note that partitioning oracles may output more than two possible outputs, for example if there are multiple distinguishable error messages, following [17].

3.2 Multi-Key Contingent Forgeries

Central to launching a partitioning oracle attack is the ability to craft splitting ciphertexts. This is formalised in the notion of "Targeted Multi-Key Contingent Forgeries", which quantifies an adversary's advantage in crafting splitting ciphertexts against a particular AEAD scheme, with oracle access to encryption. Our definition is a slight generalisation of the "Targeted Multi-Key Collision" notion from [31]; their notion can be obtained from ours by removing the adversary's encryption oracle.[4]

Targeted multi-key contingent forgery resistance (TMKCR) security is defined by the game given in Figure 1 (left). It is parameterised by a scheme AEAD and a target key set $\mathbb{K}^* \subseteq K$. A possibly randomised adversary $\mathcal{A}$ is given input a target set $\mathbb{K}^*$ and must produce nonce N^*, associated data D^* and ciphertext C^* such that $\mathsf{AuthDec}_k(N^*, D^*, C^*) \neq \perp$ for all $k \in \mathbb{K}^*$. We define the advantage via

$$\mathsf{Adv}^{\text{tmk-cr}}_{\mathsf{AEAD},\mathbb{K}^*}(\mathcal{A}) = \Pr\left[\text{TMKCR}^{\mathcal{A}}_{\mathsf{AEAD},\mathbb{K}^*} \Rightarrow 1\right] \tag{2}$$

where "$\text{TMKCR}^{\mathcal{A}}_{\mathsf{AEAD},\mathbb{K}^*} \Rightarrow 1$" denotes the event that $\mathcal{A}$ succeeds in finding N^*, D^*, 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 contingent forgery resistance goal, called $\text{MKCR}^{\mathcal{A}}_{\mathsf{AEAD},\kappa}$. The associated security game, given in Figure 1 (right), 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 D^*, and ciphertext C^*. The adversary wins if $|\mathbb{K}^*| \geq \kappa$ for some parameter $\kappa > 1$ and decryption of N^*, D^*, C^* succeeds for all $k \in \mathbb{K}^*$. We define the advantage via

$$\mathsf{Adv}^{\text{mk-cr}}_{\mathsf{AEAD},\kappa}(\mathcal{A}) = \Pr\left[\text{MKCR}^{\mathcal{A}}_{\mathsf{AEAD},\kappa} \Rightarrow 1\right] \tag{3}$$

where "$\text{MKCR}^{\mathcal{A}}_{\mathsf{AEAD},\kappa} \Rightarrow 1$" denotes the event that $\mathcal{A}$ succeeds in finding $\mathbb{K}^*$ and N^*, D^*, C^* that decrypt under all keys in $\mathbb{K}^*$. The event is defined over the coins used by $\mathcal{A}$.

[4] We hope the reader forgives our abuse of nomenclature; although we refer to both notions as TMKCR, ours is a (slight) generalisation of Len et al.'s, and we use the term "key contingent forgery" to encompass both.

Game $\mathrm{TMKCR}^{\mathcal{A}}_{\mathsf{AEAD},\mathbb{K}^*}$
00 require $\mathbb{K}^* \subset K$
01 $k \leftarrow_{\$} \mathbb{K}^*, \mathcal{N} \leftarrow \emptyset$
02 $(N^*, D^*, C^*) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{Enc}}}(\mathbb{K}^*)$
03 stop with $[\mathsf{AuthDec}_k(N^*, D^*, C^*) \neq \bot]$

Oracle $\mathcal{O}_{\mathrm{Enc}}(N, D, P)$
04 require $N \notin \mathcal{N}$
05 $\mathcal{N} \xleftarrow{\cup} N$
06 return $\mathsf{AuthEnc}_k(N, D, P)$

Game $\mathrm{MKCR}^{\mathcal{A}}_{\mathsf{AEAD},\kappa}$
00 $\mathbb{K}^* \leftarrow_{\$} \mathcal{A}(\kappa)$; require $\mathbb{K}^* \subset K$ and $|\mathbb{K}^*| \geq \kappa$
01 $k \leftarrow_{\$} \mathbb{K}^*, \mathcal{N} \leftarrow \emptyset$
02 $(N^*, D^*, C^*) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathrm{Enc}}}(\mathbb{K}^*)$
03 stop with $[\mathsf{AuthDec}_k(N^*, D^*, C^*) \neq \bot]$

Oracle $\mathcal{O}_{\mathrm{Enc}}(N, D, P)$
04 require $N \notin \mathcal{N}$
05 $\mathcal{N} \xleftarrow{\cup} N$
06 return $\mathsf{AuthEnc}_k(N, D, P)$

Fig. 1. Games modelling (targeted) multi-key contingent forgery resistance for an AEAD scheme. Note that in both cases, an adversary that can produce a ciphertext C^* that decrypts under every key in $\mathbb{K}^*$ will win the game with probability 1. **Left:** Targeted Multi-Key Collision Resistance. **Right:** Multi-key contingent forgery resistance, a weaker notion which lets the adversary choose the set of target keys $\mathbb{K}^*$.

4 Partitioning Oracle Attacks from Weak Key Forgeries

At a high level, our attack works as follows: Construct key-contingent forgeries from captured ciphertexts using weak-key forgery techniques and submit these to a decryption oracle; that is, an oracle that reveals whether a ciphertext is accepted or rejected. The weak key forgery ensures that the ciphertext will only be accepted if the user's key is in the set of weak keys.

More specifically: (1) In an offline phase, the adversary pre-computes a set of ciphertext masks. Each mask corresponds to a set of passwords to be tested. (2) In an online phase, the adversary intercepts a ciphertext and, using a ciphertext mask, constructs a key-contingent forgery which it forwards to the partitioning oracle. Observing whether or not the key-contingent forgery is accepted reveals whether or not the user's key is in the set of target keys corresponding to the ciphertext mask. Our attack relies on the ability of the adversary to act as a "machine-in-the-middle" between sender and receiver. We first give an abstract description of a key contingent forgery consisting of ℓ ciphertext blocks which encompasses two special cases: a targeted key-contingent forgery testing ℓ keys, (Section 4.1); and a targeted forgery passing format requirements on underlying plaintexts, (Section 4.2).

1. **Offline phase.** The attack takes a set of target keys $\mathbb{K}^* = \{K_1, \ldots, K_{\ell-1}\}$ as input and outputs a ciphertext mask. We note that one key is lost per ciphertext block that is not a free variable.
   (a) First derive the associated authentication (GHASH) keys by setting $\mathbb{H}^* = \{\mathsf{E}_k(0^{128}) | k \in \mathbb{K}^*\}$.
   (b) Set $q(x) = \sum_{i=1}^{\ell} q_i \cdot x^{\ell+1-i} = x \cdot \prod_{H \in \mathbb{H}^*} (x \oplus H)$.
2. **Online phase.** The online phase takes as further input a valid nonce, ciphertext, tag tuple (N, C, T) and outputs a key-contingent forgery consisting of

tuple $(N, \hat{C}, T)$. The key-contingent forgery is forwarded to the partitioning oracle. In what follows, we assume that $\ell - 1 \geq p = \lceil \mathrm{len}(C/128) \rceil$

(a) First parse the captured ciphertext as $C = C_1 \parallel \cdots \parallel C_p^*$, i. e., as blocks of the appropriate length. Let $\alpha = \mathrm{len}(C) \oplus \mathrm{len}(\hat{C})$ and β be constants. Now set $q'(x) = \sum_{i=1}^{\ell+1} q'_i \cdot x^{\ell+2-i} = (a \oplus bx) \cdot q(x)$, with

$$a = \alpha \cdot q_\ell^{-1} \text{ and } b = \beta \cdot q_2^{-1} \oplus \alpha \cdot q_1 \cdot q_2^{-1} \cdot q_\ell^{-1}. \tag{4}$$

Set $Q' = q'_1 \parallel \cdots \parallel q'_\ell$. Note that $q'_{\ell+1} = q_\ell \cdot a = \alpha$ and $q'_1 = a \cdot q_1 \oplus b \cdot q_2 = \beta$. This step can take place offline if $\mathrm{len}(C)$ is known in advance.

(b) Let $\hat{C} = C^* \oplus Q'$, where $C^* = 0^{128} \parallel \cdots \parallel 0^{128} \parallel C_1 \parallel \cdots \parallel C_p^*$ denotes the ciphertext C padded (pre-pended) with blocks of zeros to match the length of Q'. As $\ell \geq p + 1$, at least one block of padding is pre-pended. Note that if the user key $k \in \mathbb{K}^* \cup \{K_\ell\} \cup \{0\}$, where $K_\ell = a \cdot b^{-1}$, then for $H = \mathsf{E}_k(0^{128})$,

$$\begin{aligned} h_H(\hat{C}) &= \mathrm{len}(\hat{C}) \cdot H \oplus \hat{C}_\ell^* \cdot H^2 \oplus \hat{C}_{\ell-1} \cdot H^3 \oplus \cdots \oplus \hat{C}_1 \cdot H^{\ell+1} \\ &= (\alpha \oplus \mathrm{len}(C)) \cdot H \oplus \left(C_p^* \oplus q'_\ell\right) \cdot H^2 \oplus \cdots \oplus (0^{128} \oplus q'_1) \cdot H^{\ell+1} \\ &= q'(H) \oplus h_H(C). \end{aligned}$$

Consequently, the tag is a valid forgery and $\mathsf{AuthEnc}_k(N, \varepsilon, \hat{C}) \neq \perp$.

4.1 Targeted Key Contingent Forgery Testing ℓ keys

We first consider key contingent ciphertext forgeries that test ℓ keys with no restrictions on the format of the underlying plaintext. Setting $\beta = a \cdot q_2 \cdot H_\ell^{-1} \oplus a \cdot q_1$ in eq. (4) for $H_\ell = \mathsf{E}_{K_\ell}(0^{128})$ gives $a = b \cdot H_\ell$. Thus,

$$q'(x) = b \cdot (x + H_\ell) \cdot x \cdot \prod_{H \in \mathbb{H}^*} (x \oplus H) = b \cdot x \cdot \prod_{H \in \mathbb{H}^* \cup \{H_\ell\}} (x \oplus H)\,.$$

The ciphertext forgery $\hat{C}$ is a valid forgery if $k \in \mathbb{K}^* \cup K_\ell \cup \{0\}$. Thus, we are in effect able to test target key sets of size $|\mathbb{K}^*| + 1 = \ell$.

Performance. The attack description above is for a fixed set of target keys $\mathbb{K}^*$; in practice, an attacker would prepare a collection of ciphertext masks corresponding to disjoint target key sets $\{\mathbb{K}_i^*\}_{i \in I}$, such that $p_{i+1} \geq p_i$ for all i, where p_i denotes the aggregate success probability of target key set $\mathbb{K}_i^*$. Given $n = |\mathbb{K}^*|$ hashing keys, the coefficients of the polynomial $q(x)$ can be computed using $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ space. We note that the offline phase need only occur once, allowing the adversary to amortise the upfront cost of pre-computation over multiple targets. This is especially useful in cases where generating target keys from passwords is particularly slow.

In the online phase, splitting ciphertexts are then submitted in order until a query is successful; we note that a negative result is returned immediately. For a successful query, we know that the key $k \in \mathbb{K}_i^*$ for some particular i.

As our result relies on pre-computation to be practical, in order to perform a binary search on $\mathbb{K}_i^*$ appropriate forgery masks would have to be pre-computed – this would require $\mathcal{O}(n \log n)$ space. In most cases it is probably more efficient, once an adversary knows that $k \in \mathbb{K}_i^*$, to perform the first few iterations of a binary search (having precomputed the necessary values) before switching to trial decryption of C with each key in $\mathbb{K}_i^*$. We assume that the cost of querying a ciphertext is low and that either (1) there is a steady supply of ciphertexts to intercept or (2) it is possible to reuse the same nonce – the server may or may not enforce unique nonces depending on context. Regarding point (1), we note that a common adversarial model introduced by the BEAST attack [19] gives an attacker the ability to inject arbitrary plaintexts via client-side JavaScript in some window in the user's browser (see e. g. [5,6,15]).

Our attack is limited to scenarios where keys are deterministically derived from passwords; that is, if passwords are salted (using randomly generated salts) then pre-computation is no longer feasible. This highlights the fact that whilst salts are not secret values, they should be unpredictable when used to derive encryption keys from passwords, in a direct analogue to password storage. Better security in any case is obtained by using password authenticated key exchange protocols such as [26], rather than deriving session keys statically from passwords.

4.2 Targeted Key Contingent Forgery Passing Format Checks

The targeted multi-key contingent forgery attack from the previous section results in ciphertexts that decrypt under the user's key to plaintexts that are "garbage". This is a problem in cases where plaintexts are required to meet some format check. The most common form of format check will be a header field containing (for example) protocol data, sender and receiver addresses, serial numbers or integrity check values. The weak key forgery method of [36] allows full control over the underlying plaintext, with the caveat that the ciphertext forgery represents an (untargeted) multi-key contingent forgery – for every block of underlying plaintext that is part of the format check, the number of *targeted* keys being tested will decrease by one, with one extra untargeted key gained. In practice this will not make much difference: usually, the prefix is designed to be as short as possible, which means one or at most two blocks. We would typically expect splitting ciphertexts of degree ≈500 so that losing one or two blocks represents only a small fraction of the total.

Let us assume that the captured nonce, ciphertext, tag tuple (N, C, T) corresponds to some underlying plaintext matching the (known) required format. For concreteness, assume that the first block of plaintext (respectively, ciphertext) corresponds to the format to be checked. This means that we need to leave the first block of plaintext unchanged. We thus set $\beta = C_1 \oplus 0$ in eq. (4) and note that the method may easily be adapted to "flip bits" in the underlying plaintext by using a suitable value of $\beta = C_1 \oplus \delta$; furthermore, it is straightforward to extend the method to deal with multiple blocks. By construction, $\hat{C}_1 = q'_1 = \beta$, which gives $\hat{C} = C_1 \parallel \hat{C}_2 \parallel \cdots \parallel \hat{C}_\ell$, i. e., a ciphertext forgery $\hat{C}$ with the same first

block of ciphertext (and thus underlying plaintext) as the original intercepted ciphertext C. Note that we gain $K_\ell = a \cdot b^{-1}$ as an untargeted key.

Len et al. [31] show how to craft (untargeted) multi-key collisions to pass format checks with fixed prefixes, however their method is impractical for prefixes longer than a couple of bytes; in contrast, our method can easily be applied to arbitary prefixes and is targeted. Lastly, we observe that this method circumvents the key committing "padding fix" discussed in Section 2.3, i. e., to prepend a constant string to messages before encrypting. The ability to control underlying plaintexts in this way allows an attacker to apply partitioning oracle attacks using weak key forgeries where attacks based on exploiting non-committment are infeasible.

5 Partitioning Oracle Attacks against Shadowsocks

Originally written by a pseudonymous developer, Shadowsocks [1] is an encrypted proxy for TCP and UDP traffic, based on SOCKS5. Shadowsocks was first built to help evade censorship in China, and it underlies other tools such as Jigsaw's Outline VPN. To use Shadowsocks, a user first deploys the Shadowsocks proxy server on a remote machine, provisions it with a static password and chooses an encryption scheme to use for all connections. The most up-to-date implementations only support AEAD schemes for encryption, with the options consisting of AES-GCM (128-bit or 256-bit) or ChaCha20/Poly1305. Next the user configures the Shadowsocks client on their local machine, and can then forward TCP or UDP traffic from their machine to the Shadowsocks proxy server.

Len et al. [31] showed how to build a practical partitioning oracle attack against Shadowsocks proxy servers. At a high level, their attack exploited the non-key committing property of the AEAD schemes used, making it possible to craft ciphertexts which decrypt correctly under a set of target keys. Furthermore, the attack exploits the fact that the proxy server opens an ephemeral UDP port in response to a valid request (and otherwise does not) which reveals whether a ciphertext has been accepted or rejected. The attack depends on a particular configuration: password derived keys and UDP traffic. As a response to [31], users are advised to mitigate against the attack by generating good quality passwords and disabling UDP mode [7]. In this section, we first describe the Shadowsocks protocol and the partitioning oracle attack of Len et al., before going on to describe how weak key forgeries can be used to launch a partitioning oracle attack. We note that whilst our attack is rendered impractical by the per-message salt used in the Shadowsocks protocol, a description of a hypothetical attack still offers a useful case study, which we describe below.

The Shadowsocks Protocol. The client starts by hashing the user password pw to obtain a key $k = h(pw)$. The client then samples a random sixteen-byte salt s and computes a session key $k_s \leftarrow \text{HKDF}(k, s, \mathit{info})$ using HKDF [29], where *info* is the string `ss-subkey`. A new salt and session key are generated for every message. The client encrypts its plaintext payload P by computing $C \leftarrow \mathsf{AuthEnc}_{k_s}(Z, \varepsilon, \mathit{flag} \parallel \mathit{ip} \parallel \mathit{port} \parallel \mathit{payload})$ where Z denotes a nonce set to a

string of zero bytes (12 for AES-GCM); the value ε empty associated data; and *flag* is a one-byte header indicating the format of *ip* with the following convention: *flag* = 01 indicates that *ip* is a 4-byte IPv4 address, *flag* = 03 indicates that *ip* consists of a one byte length and then hostname, and *flag* = 04 indicates that *ip* is a 16-byte IPv6 address. The port field *port* is two bytes long. The client sends (s, C) to the server via UDP. If the client is using TCP, the process is the same except that the ciphertext is prefixed with a two-byte encrypted length (and authentication tag) before being sent to the server via TCP.

When the Shadowsocks server receives (s, 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. If decryption succeeds, the plaintext's format is checked by verifying that its first byte is equal to a valid *flag* value. If that check passes, the next bytes are interpreted as an appropriately encoded address *ip*, and two-byte port number *port*. Finally, the rest of the payload is sent to the remote server identified by *ip* and *port*. The proxy then listens on an ephemeral source UDP port assigned by the kernel networking stack for a reply from the remote server. 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.

The Attack of Len et al. The proxy server opens an ephemeral UDP port in repsonse to a valid request (and otherwise not). One can view this as a remotely observable logical side-channel that reveals whether decryption succeeds. The attacker starts with knowledge of a password dictionary $\mathcal{D}$ and an estimate $\hat{p}$ of the probability distribution over keys in the dictionary. The attack has two steps, a pre-computation phase and an active querying phase.

In the pre-computation phase, the attacker chooses an arbitrary salt s and derives a set of session keys $\mathbb{K} = \mathbb{K}(\mathcal{D})$ by $k_s^i \leftarrow \text{HKDF}(h(pw_i), s, \texttt{ss-subkey})$ for all $pw_i \in \mathcal{D}$; the nonce is set as a string of all zeroes. The adversary then outputs a ciphertext $\hat{C}$ of length 4093 (to meet the length restriction imposed by Shadowsocks servers) and a set $\mathbb{K}^*$ of 4091 keys such that $\hat{C}$ decrypts under every key in S to give a plaintext with first byte 01. We gloss over the details of how $\hat{C}$ is constructed and refer the reader to [31]; we note that the construction is not a targeted multi-key collision.

In the querying phase, the attacker then submits (s^*, C^*) to the proxy server. Should the user's key be in the set of target keys, $k(pw) \in \mathbb{K}^*$, 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 and open a UDP source port to listen for a response, which the attacker can observe by port scanning.

5.1 Partitioning Oracles from Weak Key Forgeries.

We now describe how to launch a partitioning oracle attack using weak key forgeries against Shadowsocks (in the same configuration as the attack of Len et al. described above). As noted above, our attack is impractical as session keys are salted on a per-message basis in the Shadowsocks protocol, making precomputation of forgery masks infeasible. Nevertheless, a weak key forgery partitioning oracle attack against Shadowsocks is an instructive case study, demonstrating the feasibility of the approach and allowing us to point out some interesting features; in particular, we are able to construct targeted multi-key contingent forgeries that meet arbitrary format requirements as we explain below.

Basic version. We separate the attack into two steps, a computation phase and an active querying phase. The attacker starts with knowledge of a password dictionary $\mathcal{D}$ and an estimate $\hat{p}$ of the probability distribution over keys in the dictionary and then intercepts a salt, ciphertext tuple (s, C).

In the computation phase, the attacker first chooses a set of passwords $\mathcal{D}^*$ with $|\mathcal{D}^*| = 4092$, such that the set has the maximum aggregate probability according to $\hat{p}$. The attacker then derives a set of session keys $\mathbb{K}^*$ from the salt s and set of passwords $\mathcal{D}^*$ by $k_s^i \leftarrow \text{HKDF}(h(pw_i), s, \texttt{ss-subkey})$; the nonce is set as a string of all zeroes. Using the weak key forgery method described in Section 4.2, the attacker outputs a ciphertext $\hat{C}$ of length 4093 (to meet the length restriction imposed by Shadowsocks servers) such that $\hat{C}$ decrypts under the users key k if $k \in \mathbb{K}^*$. Furthermore, the underlying plaintext $P \leftarrow \mathsf{AuthDec}_k(\hat{C})$ passes the format check.

In the querying phase, the attacker then submits (s, C^*) to the proxy server. Should the user's key be in the set of target keys, the server will interpret the decrypted plaintext as *flag* $\|$ *ip* $\|$ *port* $\|$ *payload*; that is, an IP address, destination port and payload. Note that these are unchanged from the original plaintext that was sent by the user, so will be accepted by the server's network protocol stack. The server will send the payload as a UDP packet and open a UDP source port to listen for a response, which the attacker can observe by port scanning.

Extension 1: Redirection (known plaintext attack). If the attacker knows the first 7 bytes of an underlying plaintext, which we write as *prefix*, then they can use the weak key forgery technique to redirect the user's payload to arbitrary destinations. In particular, the first 7 bytes can be modified to give 01 $\|$ *ip*$'$ $\|$ *port*$'$, with *ip*$'$ a four-byte IPv4 address, and *port*$'$ a two-byte destination port. This is the idea behind Peng's "redirect attack" [22,34], discovered in February 2020, which exploited the use of stream ciphers without integrity protection in the Shadowsocks protocol. Obtaining plaintexts with known *prefix* is relatively easy in the server to client direction, as many common server protocols start with the same bytes (e. g. `HTTP/1.` for HTTP). In the client to server direction, underlying plaintexts will be in the format [destination][payload], so that the adversary needs to know the target address (and its encoding), perhaps through injecting plaintexts via client-side JavaScript in some window in the user's browser [5,6,15,19]. Note that if an adversary is able to launch cho-

sen plaintext attacks, they could target the TCP configuration of Shadowsocks (the recommended option) by crafting plaintexts with the maximum length to overcome the fact that for TCP the length is sent encrypted together with the encrypted payload.

The adversary intercepts a ciphertext C from server to client, and using weak key forgery techniques modifies C to give a splitting ciphertext $\hat{C}$ whose underlying plaintext begins with $\mathit{prefix}' = 01 \parallel \mathit{ip}' \parallel \mathit{port}'$, i. e., an address that the adversary controls. The splitting ciphertext is then sent to the Shadowsocks server: if the splitting ciphertext is accepted, the payload is sent to the adversary, revealing that the user's key is in the set of target keys associated to $\hat{C}$. To produce $\hat{C}$, we modify the basic attack above as follows: when it comes to constructing the weak key forgery mask, following the technique outlined in Section 4.2, we use a non-zero value of β in Equation (4); specifically, $\beta = \mathit{prefix} \oplus (01 \parallel \mathit{ip}' \parallel \mathit{port}')$, interpreted as an element of $\mathbb{F}_{2^{128}}$. The effect is to flip some bits in the 7-byte prefix *prefix*, so that we obtain the attacker's address prefix'.

We note that this attack allows the adversary to efficiently and reliably determine whether the ciphertext has been accepted; it is no longer necessary to scan the server for open ports, which is time consuming and not necessarily completely reliable. Furthermore, if the splitting ciphertext is accepted, the adversary receives the payload *payload* which means that it can efficiently test target keys against the ciphertext by encrypting one block of plaintext and checking whether it matches. Without this, the adversary would need to calculate the authentication tag of the captured ciphertext for each target key.

Extension 2: Bypassing the padding fix. As discussed in Section 2.3, prior work on non-key committing AEAD schemes showed that applying a "padding fix", that is prepending a fixed constant string to underlying plaintexts, transforms the scheme to be key-committing. Applying a padding fix is recommended by Len et al. as a way to mitigate against partitioning oracle attacks; however, a partitioning oracle attack using weak key forgeries will still be successful despite that mitigation. To see this, we simply modify the description of the "basic attack version" in the previous subsection to leave one further block unaltered, at the cost of testing one less key per ciphertext $\hat{C}$. We note that the reason that our attack impractical is due to the salting of passwords to derive per-message ephemeral keys, rather than because of the non-key committing property of the AEAD scheme used.

5.2 Other Proxy Servers (VPNs)

Virtual Private Networks (VPNs) are often used to achieve similar objectives to Shadowsocks (allowing a user to access the internet via a proxy server), although Shadowsocks was designed specifically to circumvent internet censorship, which is not part of the threat model for VPNs. VPNs allow users to interact with what appears to be a private network, despite the interaction taking place over a public network (typically, the internet). This is achieved by encrypting packets in transit so that the contents are hidden from the public network. VPNs have a number

of applications, including enabling users to remotely access local resources, or allowing individuals to improve their anonymity and privacy online (by masking their IP and hiding their traffic). Users connect to a proxy server via an encrypted tunnel, and the proxy server acts as an intermediary for the client and the internet (or a portion thereof). The most widely used protocols for VPNs are TLS and the IPsec protocol

At a high level, IPsec works as follows: the user first composes a TLS packet that will be sent to the end destination. This is encapsulated in an IPsec Encapsulating Security Payload (ESP) packet in tunnelling mode, which essentially adds a header and encrypts the whole packet to give a ciphertext C. This encrypted packet C is sent to the proxy server, where it is decrypted to recover the underlying TLS packet. The proxy server now forwards the TLS packet to its intended destination. There are many configuration options for how the user and proxy server authenticate and/or encrypt the ESP packets, including to provision the user and proxy server with static keys [28]. This is known as "manual management", and is suited to small static environments. However, the standard does not allow AES-GCM (or ChaCha20-Poly1305) with manual keys, although they are available in other configurations, due to concerns over the brittleness when a nonce/key combination is reused. AES with HMAC is preferred, which happens to be both key-committing and not vulnerable to weak key forgeries. Similarly, OpenVPN disallows AEAD cipher mode with static keys to avoid the insecurity of potential nonce/key reuse. We have thus not been able to find any vulnerable applications "in the wild", but note that partitioning oracle attacks are theoretically possible against implementations incorrectly deviating from the specification.

6 Conclusions

Prior work demonstrated that key commitment is an important security property of AEAD schemes. Our results suggest that resistance to weak key forgeries should be considered a related design goal to key-commitment, particularly in settings that are vulnerable to partitioning oracle attacks. Concretely, our results demonstrate – in contrast to the suggestions of prior work – that structured underlying plaintexts (e.g. packet headers that prefix every plaintext message, or an appended block of all zeros) is not a sufficient mitigation against partitioning oracle attacks. Lastly, our results reinforce the message that weak passwords should never be used to derive encryption keys.

Acknowledgements

This research was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1).

The authors would like to thank Kenny Paterson for the discussion and feedback on an early draft, as well as the anonymous reviewers.

References

1. Shadowsocks - a fast tunnel proxy that helps you bypass firewalls. https://shadowsocks.org, retrieved May 2021.
2. Abdelraheem, M.A., Beelen, P., Bogdanov, A., Tischhauser, E.: Twisted polynomials and forgery attacks on GCM. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology – EUROCRYPT 2015, Part I. Lecture Notes in Computer Science, vol. 9056, pp. 762–786. Springer, Heidelberg (Apr 2015). https://doi.org/10.1007/978-3-662-46800-5_29
3. Albertini, A., Duong, T., Gueron, S., Kölbl, S., Luykx, A., Schmieg, S.: How to abuse and fix authenticated encryption without key commitment. Cryptology ePrint Archive, Report 2020/1456 (2020), https://eprint.iacr.org/2020/1456
4. Albrecht, M.R., Paterson, K.G.: Lucky microseconds: A timing attack on amazon's s2n implementation of TLS. In: Fischlin, M., Coron, J.S. (eds.) Advances in Cryptology – EUROCRYPT 2016, Part I. Lecture Notes in Computer Science, vol. 9665, pp. 622–643. Springer, Heidelberg (May 2016). https://doi.org/10.1007/978-3-662-49890-3_24
5. AlFardan, N.J., Bernstein, D.J., Paterson, K.G., Poettering, B., Schuldt, J.C.N.: On the security of RC4 in TLS. In: King, S.T. (ed.) USENIX Security 2013: 22nd USENIX Security Symposium. pp. 305–320. USENIX Association (Aug 2013)
6. AlFardan, N.J., Paterson, K.G.: Lucky thirteen: Breaking the TLS and DTLS record protocols. In: 2013 IEEE Symposium on Security and Privacy. pp. 526–540. IEEE Computer Society Press (May 2013). https://doi.org/10.1109/SP.2013.42
7. Anonymous, Anonymous, Anonymous, Fifield, D., Houmansadr, A.: A practical guide to defend against the GFW's latest active probing. https://gfw.report/blog/ss_advise/en/ (2021), retrieved May 2021.
8. Armour, M., Poettering, B.: Substitution attacks against message authentication. IACR Transactions on Symmetric Cryptology **2019**(3), 152–168 (2019). https://doi.org/10.13154/tosc.v2019.i3.152-168
9. Armour, M., Poettering, B.: Subverting decryption in AEAD. In: Albrecht, M. (ed.) 17th IMA International Conference on Cryptography and Coding. Lecture Notes in Computer Science, vol. 11929, pp. 22–41. Springer, Heidelberg (Dec 2019). https://doi.org/10.1007/978-3-030-35199-1_2
10. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Koblitz, N. (ed.) Advances in Cryptology – CRYPTO'96. Lecture Notes in Computer Science, vol. 1109, pp. 1–15. Springer, Heidelberg (Aug 1996). https://doi.org/10.1007/3-540-68697-5_1
11. Bellovin, S.M., Merritt, M.: Encrypted key exchange: Password-based protocols secure against dictionary attacks. In: 1992 IEEE Symposium on Security and Privacy. pp. 72–84. IEEE Computer Society Press (May 1992). https://doi.org/10.1109/RISP.1992.213269
12. Bernstein, D.J.: The poly1305-AES message-authentication code. In: Gilbert, H., Handschuh, H. (eds.) Fast Software Encryption – FSE 2005. Lecture Notes in Computer Science, vol. 3557, pp. 32–49. Springer, Heidelberg (Feb 2005). https://doi.org/10.1007/11502760_3
13. Bernstein, D.J.: ChaCha, a variant of Salsa20. In: Workshop record of SASC. vol. 8, pp. 3–5 (2008)
14. Beznazwy, J., Houmansadr, A.: How China detects and blocks shadowsocks. In: Proceedings of the ACM Internet Measurement Conference. pp. 111–124 (2020)

15. Bhargavan, K., Leurent, G.: On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016: 23rd Conference on Computer and Communications Security. pp. 456–467. ACM Press (Oct 2016). https://doi.org/10.1145/2976749.2978423
16. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In: Krawczyk, H. (ed.) Advances in Cryptology – CRYPTO'98. Lecture Notes in Computer Science, vol. 1462, pp. 1–12. Springer, Heidelberg (Aug 1998). https://doi.org/10.1007/BFb0055716
17. Boldyreva, A., Degabriele, J.P., Paterson, K.G., Stam, M.: On symmetric encryption with distinguishable decryption failures. In: Moriai, S. (ed.) Fast Software Encryption – FSE 2013. Lecture Notes in Computer Science, vol. 8424, pp. 367–390. Springer, Heidelberg (Mar 2014). https://doi.org/10.1007/978-3-662-43933-3_19
18. Dodis, Y., Grubbs, P., Ristenpart, T., Woodage, J.: Fast message franking: From invisible salamanders to encryptment. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018, Part I. Lecture Notes in Computer Science, vol. 10991, pp. 155–186. Springer, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96884-1_6
19. Duong, T., Rizzo, J.: Here come the ⊕ ninjas. Unpublished manuscript, `https://tlseminar.github.io/docs/beast.pdf`, retrieved May 2021.
20. Dworkin, M.J.: SP 800-38D. recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. Tech. rep. (2007)
21. Farshim, P., Orlandi, C., Roşie, R.: Security of symmetric primitives under incorrect usage of keys. IACR Transactions on Symmetric Cryptology **2017**(1), 449–473 (2017). https://doi.org/10.13154/tosc.v2017.i1.449-473
22. Fifield, D.: Decryption vulnerability in shadowsocks stream ciphers. `https://github.com/net4people/bbs/issues/24`, retrieved May 2021.
23. Garman, C., Green, M., Kaptchuk, G., Miers, I., Rushanan, M.: Dancing on the lip of the volcano: Chosen ciphertext attacks on apple iMessage. In: Holz, T., Savage, S. (eds.) USENIX Security 2016: 25th USENIX Security Symposium. pp. 655–672. USENIX Association (Aug 2016)
24. Grubbs, P., Lu, J., Ristenpart, T.: Message franking via committing authenticated encryption. In: Katz, J., Shacham, H. (eds.) Advances in Cryptology – CRYPTO 2017, Part III. Lecture Notes in Computer Science, vol. 10403, pp. 66–97. Springer, Heidelberg (Aug 2017). https://doi.org/10.1007/978-3-319-63697-9_3
25. Handschuh, H., Preneel, B.: Key-recovery attacks on universal hash function based MAC algorithms. In: Wagner, D. (ed.) Advances in Cryptology – CRYPTO 2008. Lecture Notes in Computer Science, vol. 5157, pp. 144–161. Springer, Heidelberg (Aug 2008). https://doi.org/10.1007/978-3-540-85174-5_9
26. Jarecki, S., Krawczyk, H., Xu, J.: OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In: Nielsen, J.B., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2018, Part III. Lecture Notes in Computer Science, vol. 10822, pp. 456–486. Springer, Heidelberg (Apr / May 2018). https://doi.org/10.1007/978-3-319-78372-7_15
27. Joux, A.: Authentication failures in NIST version of GCM. Tech. rep. (2006)
28. Kent, S., Seo, K.: Security architecture for the internet protocol. RFC 4301, RFC Editor (December 2005), `https://tools.ietf.org/html/rfc4301`
29. Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: Rabin, T. (ed.) Advances in Cryptology – CRYPTO 2010. Lecture Notes in Computer Science, vol. 6223, pp. 631–648. Springer, Heidelberg (Aug 2010). https://doi.org/10.1007/978-3-642-14623-7_34

30. Krawczyk, H.: The opaque asymmetric PAKE protocol (draft). Tech. rep. (2018), https://datatracker.ietf.org/doc/html/draft-krawczyk-cfrg-opaque-02
31. Len, J., Grubbs, P., Ristenpart, T.: Partitioning oracle attacks. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 195–212. USENIX Association (Aug 2021), https://www.usenix.org/conference/usenixsecurity21/presentation/len
32. McGrew, D., Viega, J.: The galois/counter mode of operation (GCM). Tech. rep. (2004), http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-revised-spec.pdf
33. Pal, B., Daniel, T., Chatterjee, R., Ristenpart, T.: Beyond credential stuffing: Password similarity models using neural networks. In: 2019 IEEE Symposium on Security and Privacy. pp. 417–434. IEEE Computer Society Press (May 2019). https://doi.org/10.1109/SP.2019.00056
34. Peng, Z.: Redirect attack on shadowsocks stream ciphers. https://github.com/edwardz246003/shadowsocks, retrieved May 2020.
35. Procter, G.: A security analysis of the composition of ChaCha20 and Poly1305. Cryptology ePrint Archive, Report 2014/613 (2014), https://eprint.iacr.org/2014/613
36. Procter, G., Cid, C.: On weak keys and forgery attacks against polynomial-based MAC schemes. In: Moriai, S. (ed.) Fast Software Encryption – FSE 2013. Lecture Notes in Computer Science, vol. 8424, pp. 287–304. Springer, Heidelberg (Mar 2014). https://doi.org/10.1007/978-3-662-43933-3_15
37. Saarinen, M.J.O.: Cycling attacks on GCM, GHASH and other polynomial MACs and hashes. In: Canteaut, A. (ed.) Fast Software Encryption – FSE 2012. Lecture Notes in Computer Science, vol. 7549, pp. 216–225. Springer, Heidelberg (Mar 2012). https://doi.org/10.1007/978-3-642-34047-5_13
38. Vaudenay, S.: Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS... In: Knudsen, L.R. (ed.) Advances in Cryptology – EUROCRYPT 2002. Lecture Notes in Computer Science, vol. 2332, pp. 534–546. Springer, Heidelberg (Apr / May 2002). https://doi.org/10.1007/3-540-46035-7_35
39. Winter, P., Lindskog, S.: How the great firewall of China is blocking Tor. In: Dingledine, R., Wright, J. (eds.) 2nd USENIX Workshop on Free and Open Communications on the Internet, FOCI '12, Bellevue, WA, USA, August 6, 2012. USENIX Association (2012), https://www.usenix.org/conference/foci12/workshop-program/presentation/winter