Obfuscated Key Exchange

Felix Günther
IBM Research Europe – Zurich
Rüschlikon, Switzerland
mail@felixguenther.info

Douglas Stebila
University of Waterloo
Waterloo, Ontario, Canada
dstebila@uwaterloo.ca

Shannon Veitch
ETH Zurich
Zürich, Switzerland
shannon.veitch@inf.ethz.ch

Abstract

Censorship circumvention tools enable clients to access endpoints in a network despite the presence of a censor. Censors use a variety of techniques to identify content they wish to block, including filtering traffic patterns that are characteristic of proxy or circumvention protocols and actively probing potential proxy servers. Circumvention practitioners have developed *fully encrypted protocols* (FEPs), intended to have traffic that appears indistinguishable from random. A FEP is typically composed of a key exchange protocol to establish shared secret keys, and then a secure channel protocol to encrypt application data; both must avoid revealing to observers that an obfuscated protocol is in use.

We formalize the notion of *obfuscated key exchange*, capturing the requirement that a key exchange protocol's traffic "looks random" and that it resists active probing attacks, in addition to ensuring secure session keys and authentication. We show that the Tor network's obfs4 protocol satisfies this definition. We then show how to extend the obfs4 design to defend against stronger censorship attacks and present a quantum-safe obfuscated key exchange protocol. To instantiate our quantum-safe protocol using the ML-KEM (Kyber) standard, we present Kemeleon, a new mapping between ML-KEM public keys/ciphertexts and uniform byte strings.

1 Introduction

Internet censors in control of a network use various techniques to prevent clients from reaching destinations, including looking for particular patterns that are characteristic of proxy or circumvention protocols. To evade this class of blocking, censorship circumvention practitioners have developed fully encrypted protocols (FEPs) like obfsproxy [56], Shadowsocks [4], and VMess [3], which aim to hide any pattern by making the entire communication appear indistinguishable from random. This is in contrast to typical secure connection protocols, such as TLS [50], which have recognizable formats and plaintext fields, both in their handshake phase establishing the connection through some key exchange as well as in the secure channel phase when sending application data. These obfuscated protocols are used for a broad range of applications, from generally providing access to blocked or censored websites [3, 4, 49] to obfuscating specific filesharing protocols, such as eDonkey [1] and BitTorrent [2]. Another proposed use of FEPs is for generating random-looking transcripts in pseudorandom compact TLS [52].

Despite being a crucial component for enabling censorship-resilient communication, fully encrypted protocols have received only minimal attention from the academic community. Most of the research on FEPs focuses on detection and identification attacks. Only very recently, a preliminary study [24] analyzed the secure channel phase of FEPs to put forward security notions and constructions. Likewise, establishing shared keys and agreement on a protocol without revealing to passive observers that an obfuscated protocol is in use is key in establishing hidden connections, and yet no formal security definition for the handshake phase exists.

`obfs2..3..4`*: an iterative design process.* obfsproxy [54–56] is a prototypical fully encrypted protocol: it is the most-used method of accessing the Tor network from censored regions (~75k concurrent users among ~110k total pluggable transport users as of May 2024 [57]) and targets a sophisticated threat model founded on real behavior from censors. The inaugural obfsproxy design, `obfs2`, was insecure against passive observers who were able to decrypt handshake messages. The subsequent generation, `obfs3`, amended the vulnerability in `obfs2` by establishing secure session keys. Over time, the design sequentially adapted to new and stronger abilities of censors. For example, dating back to at least 2015, there has been evidence of governments performing active probing attacks wherein censors try to establish a connection with a server in an attempt to identify whether it is a proxy/obfuscation server [23]. The latest design of the protocol, `obfs4`, is catered to defend against such probing attacks. Yet, as a consequence of this ad-hoc design process, no formal analysis of the security guarantees of `obfs4` has been performed to date, despite `obfs4` being the state-of-the-art for obfuscated communication.

1.1 Our contributions

In this work, we bring concrete technical results to the landscape of FEPs. We will present results formalizing obfuscated key exchange, capturing these goals concretely and analyzing the latest obfsproxy design. Additionally, we show how to, in a systematic way, strengthen `obfs4`'s security against more powerful censors and make it quantum-safe. Finally, we discuss further challenges.

Obfuscated key exchange: more than just random messages. Our work focuses on capturing the handshake phase of fully encrypted protocols, which we term *obfuscated key exchange*. We introduce a game-based security model (in Section 4) that incorporates classical, Bellare–Rogaway-style [12] properties of key exchange, such as key indistinguishability and authentication, while also capturing the more sophisticated properties of threat models for FEPs, such as probing resistance and, of course, obfuscation itself.

Our model aims to capture obfuscation properties that are effective for censorship circumvention. This requires some care in defining security guarantees so that we do not impose inadequate requirements. For example, although FEPs are also known as "look-like-nothing" or "randomized" protocols, messages with excessively high levels of entropy are an identifiable feature of protocols that a censor can leverage [66]. Therefore, we define obfuscation as the guarantee that a given protocol is indistinguishable from other protocols within a certain *class* of protocols. We define such a class of protocols via a simulator that determines the traffic pattern and

	obfs family [54–56]			**Our protocols**		**Our model**
Property	obfs2	obfs3	obfs4	st-obfs	pq-obfs	ObfKE
Key indistinguishability		✓	✓	✓	✓	✓
Explicit authentication			✓	✓	✓	✓
Probing resistance			✓	✓	✓	✓
Obfuscation of …						
— packet lengths	✓	✓	✓	✓	✓	$\checkmark_{\mathcal{S}}$
— explicit header fields		✓	✓	✓	✓	$\checkmark_{\mathcal{S}}$
— entropy, directionality						$\checkmark_{\mathcal{S}}$
— timing properties						
Obfuscation when pk revealed				✓	✓	$\checkmark_{\text{opt}}$
Quantum safety					✓	$\checkmark_{\text{opt}}$

Table 1: Overview of properties of obfuscated protocols. Checkmarks in the obfs columns indicate which properties each protocol version achieves. The final column indicates properties our model captures: $\checkmark_{\mathcal{S}}$ denotes a simulator-dependent property; $\checkmark_{\text{opt}}$ an optional one.

distribution of messages in the protocol. The simulator-based definition allows us to capture properties such as packet lengths and explicit header fields (e.g., plaintext TLS fields such as extensions or ciphersuites) that are used by censors to classify traffic and obfuscated by FEPs. We can also capture entropy and directionality of packets (e.g., the number of packets sent in each direction), which are *not* obfuscated by modern FEPs, but used by censors to classify traffic. We discuss these security goals in detail when introducing our model formally (Section 4.1), and further motivate them by a review of protocol features used for classifying traffic (Section 8.2). Table 1 summarizes the properties that our model captures and those that are achieved by each protocol in the obfsproxy family and by the new protocols we propose in this work, discussed next.

Security analysis and extensions for obfs4. We then apply our security model to the obfs4 handshake (in Section 5), establishing the first formal proof of its security as an obfuscated key exchange, including probing resistance and obfuscation. As a useful ingredient, we introduce the notion of obfuscated key generation (in Section 2). This formalizes mappings of public keys to uniform random strings used in obfuscated key exchange protocols, and allows us to make precise the parameters of the Elligator2 mapping [13] (used in obfs4 for X25519 [40] Diffie–Hellman (DH) public keys), Uniform DH [30], and Telex's original-or-twist method [67].

One limitation of obfs4 is that its obfuscation property relies on semi-secret public-key information of Tor bridge servers. Specifically, these public keys are assumed to only be known by honest clients and unknown to adversaries, enabling a server to identify an honest client attempting to connect to the proxy service. By design, and confirmed by our security model, the current version of obfs4 provides obfuscation in the case where these public keys are never revealed to an adversary; however, as soon as such keys are revealed, one can identify current as well as past and future traffic as obfs4 traffic. Our model allows us to express a stronger obfuscation property for a new variant of the protocol, st-obfs, that we prove secure under the same assumptions as for obfs4: there, traffic remains obfuscated even when the public-key information that honest clients need to connect leaks to an adversary.

In a different direction, we tackle challenges in making protocols like obfs4 quantum-safe. Classically, obfs4 and other FEPs employ a Diffie–Hellman-based key exchange (e.g., ntor [31]) to establish key material, encoding the DH public keys into uniform strings on the wire. Towards constructing a quantum-safe version of obfs4, we introduce (in Section 2.4) a novel encoding algorithm, dubbed Kemeleon, for mapping ML-KEM [46] public keys and ciphertexts to random byte strings; ML-KEM, previously known as Kyber, is the post-quantum KEM based on module learning-with-errors that NIST has selected for standardization. We formalize this via the notion of an obfuscated KEM, showing that the approach taken for Kemeleon generalizes and preserves the relevant KEM security properties. This ultimately allows us to use obfuscated KEMs as a building block in a quantum-safe obfuscated protocol design, pq-obfs, which we propose and prove secure (in Section 7).

Discussion and further challenges. Finally, we discuss (in Section 8) how the guarantees provided by our security model and its simulator-based definition relate to obfuscation challenges in practice. We perform a brief literature review of works that detect obfuscated protocols, extracting relevant features used by censors to classify (obfuscated) protocol traffic and hence being in need of obfuscation. Our review suggests that censors can use not only byte distributions and packet lengths to identify protocols, but also timing information and directionality of packets. While our obfuscated key exchange model captures a wide range of these features already, our review points to further challenges for cryptographic modeling, given that timing information has been difficult to express formally.

1.2 Related Work

Covert authentication and authenticated key exchange. The problem of initiating an interaction in a covert (or steganographic) way has been considered in prior academic literature. von Ahn and Hopper [60] were the first to study public-key steganographic protocols and among them steganographic key exchange in which two parties establish a shared key through a sequence of messages indistinguishable from normal traffic on some communication channel. Focused on simple protocols (sending/receiving one message per party) and security notions (distinguishing a single, passive protocol trace), this work formalizes the idea of "looking-like-nothing" key exchange and considers a basic construction, but does not consider active attacks or authentication. Jarecki [38] studied covert authentication based on conditional KEMs and identity escrow schemes; however, did not consider concurrent security, protection against active attacks, or guaranteed independence of session keys. More recently, Eldefrawy, Genise, and Jarecki [22] proposed a more efficient covert authenticated key exchange protocol based on similar building blocks, and proven secure in the UC framework. Their work is in the group setting, where a trusted group manager issues certificates to parties who can then covertly establish a key if their groups match.

Notably, parties in the covert authentication setting do not know each others public keys. In contrast, in the obfuscated key exchange protocols like obfs4 that we consider, the server's public key is known to clients (but not to the censor), which allows the deployed real-world protocols to use simpler building blocks. These deployed protocols aim for security properties not covered in prior work on covert authenticated key exchange, such as explicit authentication, probing resistance, and stronger obfuscation when public keys are revealed, which our model captures.

Encrypted and covert TLS key exchanges. Prior work has considered how to hide plaintext fields or obfuscate additional information in TLS handshakes. Schwartz and Patton [52] propose using a pre-shared key (from a cTLS template) to encipher/decipher all not-already-encrypted values (i.e., handshake and ciphertext headers) with a tweakable strong pseudorandom permutation. They note a similar motivation to ours, stating that "every cTLS template potentially results in a distinct wire image, with important implications for user privacy," but assume a different setting in which a client and server already have an established pre-shared key. Meanwhile, Encrypted Client Hello [51] proposes a method for encrypting the plaintext Server Name Indication and other potentially sensitive fields in ClientHello messages. The draft includes a discussion of a "Do Not Stick Out" criterion [51, Section 10.9.4], that is similar to the obfuscation goals of FEPs, although specific to the case of TLS handshakes. The recently proposed *stealth key exchange* [26] analyzes how to embed a covert key into the nonce of a TLS 1.3 handshake. The approach uses similar techniques that are used in obfuscated key exchanges, such as the Elligator2 encoding [13] for mapping keys to random strings. We discuss how to apply our model to protocols that attempt to simulate a TLS 1.3 handshake (in Section 8.1).

Fully encrypted protocols. Most research on FEPs for censorship circumvention relates to classifying protocol behaviour and measuring features used by censors to block traffic [5, 6, 23, 28, 61, 66]. Although this work has proven useful for determining some necessary properties that a FEP should achieve in order to evade censorship, the protocols themselves remain without any formal security analysis. Only recently, Fenske and Johnson [24, 25] proposed a formal model and construction for the secure channel phase of FEPs (in both the datastream and datagram settings), and applied their work to evaluate existing protocols, including Shadowsocks [4]. Our work complements that of Fenske and Johnson, by providing a formal analysis of the key exchange phase of FEPs. Moreover, we conduct a literature review of techniques used to classify obfuscated protocols in Section 8.2 to better inform our security models and ensure that our goals are grounded in reasonable assumptions.

2 A Useful Ingredient: Making Public Keys and Ciphertexts Look Random

An identifying feature of many key exchange protocols is the placement and distribution of public keys in the handshake messages. Whereas padding and MAC tags can be shown to be indistinguishable from random, the same cannot be said for public keys. Therefore, obfuscated key exchange protocols must take one of two approaches: embedding public keys in the same positions that cover protocols place public keys, or encoding public keys as random (or other) strings. The latter approach is the strategy taken by `obfs4`, which uses the Elligator2 mapping [13] to encode X25519 [40] public keys as uniformly random strings. Other examples of mappings of public keys to random strings include Uniform DH [30], Telex's original-or-twist method [67], and Elligator Squared [58]. We present a systematic treatment of similar maps in Appendix B. Here, we present the first formal definitions capturing these encodings and security properties that will be useful in proving the security of our key exchange protocols. Additionally, we present a new encoding for ML-KEM [46] public keys and ciphertexts, which later acts as a building block towards a post-quantum obfuscated key encapsulation mechanism.

2.1 Obfuscating Public Keys

Often, encodings do not work for all public keys, but, for example, only for half of them. We hence deem it useful to couple the encoding process with the key generation process, as follows.

Definition 2.1 (Obfuscated key generation scheme)**.** *An obfuscated key generation scheme* $\mathsf{O} = (\mathsf{KGen}, \mathsf{Encode}, \mathsf{Decode})$ *with obfuscated key length* $\mathsf{ol} \in \mathbb{N}$ *consists of three algorithms:*

- $\underline{\mathsf{KGen}}() \xrightarrow{\$} (sk, pk, \hat{pk})$ *is the randomized* (obfuscated) key generation *algorithm that, with no input, generates a secret key sk, public key pk, and obfuscated public key* $\hat{pk} \in \{0,1\}^{\mathsf{ol}}$.
- $\underline{\mathsf{Encode}}(pk) \xrightarrow{\$} \hat{pk}$ *is the (possibly randomized)* encoding *algorithm that on input a public key pk outputs an obfuscated public key* $\hat{pk} \in \{0,1\}^{\mathsf{ol}}$ *or an error* $\perp$.
- $\underline{\mathsf{Decode}}(\hat{pk}) \to pk$ *is the deterministic* decoding *algorithm that on input an obfuscated public key* $\hat{pk} \in \{0,1\}^{\mathsf{ol}}$ *outputs a public key pk.*

For correctness, we demand that (obfuscated) public keys generated by KGen *can be successfully encoded/decoded into each other:*

$$\Pr\left[\begin{array}{c} \mathsf{Encode}(pk) = \hat{pk} \wedge \\ \mathsf{Decode}(\hat{pk}) = pk \end{array} \middle| (sk, pk, \hat{pk}) \xleftarrow{\$} \mathsf{KGen}() \right] = 1.$$

Most encodings follow what we call the "sample-and-encode" paradigm, where a regular key is sampled and then encoded into an obfuscated one, possibly repeating the process if the sampled key does not allow for an encoding (rejection sampling).

Definition 2.2 (Sample-and-encode obfuscated key generation scheme)**.** *Let* KGen′ *be a key generation algorithm and* Encode, Decode *be encoding/decoding algorithms working on the public key space of* KGen′*. We define the corresponding sample-and-encode obfuscated key generation scheme* $\mathsf{O} = (\mathsf{KGen}, \mathsf{Encode}, \mathsf{Decode})$ *as:*

```
KGen():
repeat
    $(sk, pk) \xleftarrow{\$} \mathsf{KGen}'()$
    $\hat{pk} \leftarrow \mathsf{Encode}(pk)$
until $\hat{pk} \neq \perp$
return $(sk, pk, \hat{pk})$
```

For sample-and-encode obfuscated key generation schemes, we will in security reductions be interested in the probability that the first sampled (sk, pk) pair through KGen′ can be successfully encoded (i.e., the probability that no resampling is necessary).

Definition 2.3 (First-sample success probability)**.** *Let* O *be a sample-and-encode obfuscated key generation scheme based on key generation algorithm* KGen′*. The first-sample success probability of* O *is*

$$\epsilon_{\mathsf{O}}^{\mathsf{1succ}} := \Pr[\mathsf{Encode}(pk) \neq \perp \mid (sk, pk) \xleftarrow{\$} \mathsf{KGen}'()].$$

We are also interested in obfuscated key generation schemes whose obfuscated public key is (statistically) close to uniform.

```
Elligator2.Encode($P = (u, v)$):
if $u = -A$ or $-Zu(u + A)$ is
    not a square then return $\perp$
if $v = 0$ and $u \neq 0$ then return $\perp$
if $v \geq 0$ then $r \leftarrow \sqrt{-u/(Z(u + A))}$
if $v < 0$ then $r \leftarrow \sqrt{-(u + A)/(Zu)}$
return $r$
```

```
Elligator2.Decode($r$):
$w \leftarrow -A/(1 + Zr^2)$
$e \leftarrow \texttt{legendre}(w^3 + Aw^2 + Bw)$
$u \leftarrow ew - (1 - e)(A/2)$
$v \leftarrow -e\sqrt{u^3 + Au^2 + Bu}$
return $P = (u, v)$
```

Figure 1: Elligator2 **encoding and decoding algorithms**

Definition 2.4 (Public key uniformity)**.** *Let* O *be an obfuscated key generation scheme (as per Definition 2.1). We measure the uniformity of the obfuscated public keys of length* ol *generated by* O *against an unbounded adversary* $\mathcal{A}$ *as*

$$\mathsf{Adv}_{\mathsf{O}}^{\mathsf{pk\text{-}unif}}(\mathcal{A}) := 2 \Pr\left[\mathcal{A}(\hat{pk}_b) = b \;\middle|\; \begin{array}{l} b \xleftarrow{\$} \{0,1\}, \hat{pk}_0 \xleftarrow{\$} \{0,1\}^{\mathsf{ol}}, \\ (sk_1, pk_1, \hat{pk}_1) \xleftarrow{\$} \mathsf{O.KGen}() \end{array}\right] - 1.$$

The obfs4 protocol uses the Elligator2 encoding, which maps elliptic curve public keys to random strings. When applied to X25519 [40] keys (over Curve25519), it satisfies our definition of an obfuscated key generation scheme with first-sample success probability $\approx \frac{1}{2}$ and public key uniformity $\approx 2^{-250.83}$. The complete description and analysis of Elligator2 follows.

Elligator2 *[13].* Elligator maps elliptic curve points to random strings (more specifically, to a *representative* element r in a field). The underlying finite field must have a characteristic that is close to a power of two in order to achieve good public key uniformity. Herein, we refer only to the Elligator2 construction from [13], since it is more widely applicable than Elligator1 and thus has been more widely adopted.

To define the map, we first instantiate some parameters:

- a finite field $GF(q)$ of odd characteristic where q is a prime power, e.g., $q = 2^{255} - 19$ for Curve25519;
- a curve E defined over $GF(q)$ of the form $v^2 = u^3 + Au^2 + Bu$ (parameterized by A, B);
- a non-square Z in $GF(q)$ (typically small); and
- a set of non-negative field elements S (common choices are $\{0, 1, \ldots, (q-1)/2\}$ or the set of even numbers).

The encoding and its inverse are then defined in Figure 1, where r is a representative (element) of the field $GF(q)$, and `legendre`(x) returns 1 if x is a square in $GF(q)$ and 0 otherwise.

Most implementations of Elligator are written for Curve25519 where $q = 2^{255} - 19$, $A = 486662$, and $Z = 2$. The encoding maps to the range of integers $[0, (q-1)/2]$. It follows from the fact that the decoding function is injective that the public key uniformity (cf. Definition 2.4) is:

$$\begin{aligned} \Delta(\mathcal{U}, \mathcal{Z}) &= \frac{1}{2} \sum_{\alpha \in [0, 2^{254}-1]} |\Pr[\mathcal{Z} = \alpha] - \Pr[\mathcal{U} = \alpha]| \\ &= \frac{1}{2}\left(\frac{q+1}{2} \cdot \left|\frac{2}{q+1} - \frac{1}{2^{254}}\right| + \left(2^{254} - \frac{q+1}{2}\right) \cdot \left|\frac{1}{2^{254}}\right|\right) \\ &= 1 - \frac{q+1}{2^{255}} \approx 2^{-250.83}. \end{aligned}$$

With these such parameters, the first-sample success probability (cf. Definition 2.3) is

$$\begin{aligned} &\Pr[u \neq -486662, -2u(u + 486662) \text{ is a QR mod } q, u = 0 \mid v = 0] \\ &\qquad = 1 \cdot \Pr[-2u(u + 486662) \text{ is a QR mod } q] \cdot 1 \approx 2^{-1}, \end{aligned}$$

which follows from the fact that for curves of this form, approximately half of the x-coordinates are quadratic residues modulo q.

2.2 Obfuscating KEM Public Keys and Ciphertexts

For obfuscation of Diffie–Hellman-based key exchange, it suffices to have encodings of public keys as random strings. However, in a KEM setting, ciphertexts must also be transmitted and may not follow a uniformly random byte distribution. Here we present analogous definitions of obfuscated key generation schemes and obfuscation uniformity for KEMs.

Definition 2.5 (Obfuscated KEM)**.** *An obfuscated KEM* OKEM = (KGen, Encode, Decode, Encap, Decap, EncodeCtxt, DecodeCtxt) *with obfuscated key and ciphertext lengths* ol, cl $\in \mathbb{N}$ *consists of seven algorithms and a parameter* cl $\in \mathbb{N}$*:*

- (KGen, Encode, Decode) *is an* obfuscated key generation scheme *with (key) obfuscation length* ol *(Definition 2.1).*
- $\underline{\mathsf{Encap}}(pk) \xrightarrow{\$} (c, K, \widehat{c})$ *is the randomized* (obfuscated) encapsulation *algorithm that takes as input a KEM public key pk, and outputs a ciphertext c, key K, and obfuscated ciphertext* $\widehat{c} \in \{0,1\}^{\mathsf{cl}}$*.*
- $\underline{\mathsf{Decap}}(sk, \widehat{c}) \rightarrow (K)$ *is the* (obfuscated) decapsulation *algorithm that takes as input a secret key sk and obfuscated ciphertext* $\widehat{c}$*, and outputs a key K.*
- $\underline{\mathsf{EncodeCtxt}}(c) \xrightarrow{\$} \widehat{c}$ *is the (possibly randomized)* encoding *algorithm that on input a ciphertext c outputs an obfuscated ciphertext* $\widehat{c} \in \{0,1\}^{\mathsf{cl}}$ *or an error* $\perp$*.*
- $\underline{\mathsf{DecodeCtxt}}(\widehat{c}) \rightarrow c$ *is the deterministic* decoding *algorithm that on input an obfuscated ciphertext* $\widehat{c} \in \{0,1\}^{\mathsf{cl}}$ *outputs a ciphertext c.*

For correctness of obfuscation, we need that (obfuscated) ciphertexts output by OKEM *can be successfully encoded/decoded into each other:*

$$\Pr\left[\begin{array}{c} \mathsf{DecodeCtxt}(\mathsf{EncodeCtxt}(c)) = c \\ \wedge\ \mathsf{DecodeCtxt}(\widehat{c}) = c \end{array} \;\middle|\; \begin{array}{l} (sk, pk, \widehat{pk}) \xleftarrow{\$} \mathsf{KGen}(), \\ (c, K, \widehat{c}) \xleftarrow{\$} \mathsf{Encap}(pk) \end{array}\right] = 1.$$

Definition 2.6 (OKEM encapsulation/decapsulation correctness)**.** *We say that an obfuscated KEM* OKEM *is* δ_{OKEM}-correct *if*

$$\Pr\left[\mathsf{Decap}(sk, \widehat{c}) \neq K \;\middle|\; \begin{array}{l} (sk, pk, \widehat{pk}) \xleftarrow{\$} \mathsf{KGen}(), \\ (c, K, \widehat{c}) \xleftarrow{\$} \mathsf{Encap}(pk) \end{array}\right] \leq \delta_{\mathsf{OKEM}}.$$

Definition 2.7 (OKEM public key collision probability)**.** *Let* OKEM *be an obfuscated KEM. We define the* public key collision probability *of* OKEM *for* $n \in \mathbb{N}$ *public keys as*

$$\mathsf{pkcoll}_{\mathsf{OKEM}}(n) := \Pr\left[\begin{array}{c} pk_i = pk_j \\ \wedge i \neq j \end{array} \;\middle|\; \begin{array}{c} (sk_i, pk_i, \widehat{pk}_i) \xleftarrow{\$} \mathsf{OKEM.KGen}() \\ \textit{for } i \in [1, n] \end{array}\right].$$

As with obfuscated key generation schemes, an obfuscated KEM may follow an analogous "encapsulate-and-encode" paradigm, where a ciphertext that is output from an encapsulation is encoded into an obfuscated one, possibly repeating the encapsulation process if the ciphertext does not allow for an encoding (rejection sampling).

Definition 2.8 ((Sample/)Encapsulate-and-encode obfuscated KEM). *Let* KEM′ = (KGen, Encap′, Decap′) *be a KEM, let* Encode, Decode *be encoding/decoding algorithms working on the public key space of* KGen, *and let* EncodeCtxt, DecodeCtxt *be encoding/decoding algorithms working on the ciphertext space of* Encap′. *We define the corresponding encapsulate-and-encode obfuscated KEM* OKEM = (KGen, Encode, Decode, Encap, Decap, EncodeCtxt, DecodeCtxt) *with:*

```
Encap(pk):
repeat
    $(c, K) \xleftarrow{\$} \mathsf{Encap}'(pk)$
    $\widehat{c} \xleftarrow{\$} \mathsf{EncodeCtxt}(c)$
until $\widehat{c} \neq \bot$
return $(c, K, \widehat{c})$
```

```
Decap(sk, ĉ):
$c \leftarrow \mathsf{DecodeCtxt}(\widehat{c})$
return $\mathsf{Decap}'(sk, c)$
```

Moreover, if KGen *is of type sample-and-encode (Definition 2.2), we call* OKEM *a* sample/encapsulate-and-encode *obfuscated KEM.*

Similar to sample-and-encode obfuscated key generation schemes, for encapsulate-and-encode obfuscated KEMs we are interested in: (1) the probability that the first ciphertext output from Encap′ can be successfully encoded, and (2) obfuscated ciphertexts that are statistically close to uniform.

Definition 2.9 (First-encap success probability). *Let* OKEM *be a encapsulate-and-encode obfuscated KEM based on* KEM′ = (KGen′, Encap′, Decap′). *The first-encap success probability of* OKEM *is*

$$\epsilon^{\mathsf{1encsucc}}_{\mathsf{OKEM}} := \Pr\left[\mathsf{EncodeCtxt}(c) \neq \bot \;\middle|\; \begin{array}{c} (sk, pk) \xleftarrow{\$} \mathsf{KGen}'(), \\ (c, K) \xleftarrow{\$} \mathsf{Encap}'(pk) \end{array}\right].$$

Definition 2.10 (Ciphertext uniformity). *Let* OKEM.Encap *be an obfuscated KEM (as per Definition 2.5). We measure the uniformity of the obfuscated ciphertext of length* cl *generated by* OKEM.Encap *against an unbounded adversary* $\mathcal{A}$ *as*

$$\mathsf{Adv}^{\mathsf{ctxt\text{-}unif}}_{\mathsf{OKEM}}(\mathcal{A}) :=$$

$$2 \cdot \Pr\left[\mathcal{A}(\widehat{c}_b) = b \;\middle|\; \begin{array}{c} b \xleftarrow{\$} \{0,1\}, \widehat{c}_0 \xleftarrow{\$} \{0,1\}^{\mathsf{cl}}, \\ (sk, pk, \widehat{pk}) \xleftarrow{\$} \mathsf{OKEM.KGen}(), \\ (c_1, K_1, \widehat{c}_1) \xleftarrow{\$} \mathsf{OKEM.Encap}(pk) \end{array}\right] - 1.$$

2.3 Obfuscated KEM Security

In our protocol analysis, we want to rely on standard KEM security notions, but we will be working with *obfuscated KEMs*. In the following we naturally extend both traditional KEM security notions to obfuscated KEMs: both indistinguishability under chosen-ciphertext attacks (IND-CCA) as well as strong pseudorandomness under chosen-ciphertext attacks (SPR-CCA), which was introduced and established for Kyber in [43, 68]. We then show that for sample/encapsulate-and-encode obfuscated KEMs (Definition 2.8), the two notions are implied by the underlying KEM achieving them and the obfuscated KEM's first-sample/encap success probability and ciphertext uniformity. This in particular will apply to our obfuscated KEM ML-Kemeleon which we introduce in Section 2.4.

```
$\mathsf{G}^{\mathsf{IND\text{-}CCA}}_{\mathsf{K}}(\mathcal{A})$:
$b \xleftarrow{\$} \{0,1\}$
$(pk, sk, \boxed{\hat{pk}}) \xleftarrow{\$} \mathsf{KGen}()$
$(c^*, K_1^*, \boxed{\widehat{c^*}}) \xleftarrow{\$} \mathsf{Encap}(pk)$
$K_0^* \xleftarrow{\$} \mathcal{K}$
$x \leftarrow c^* \boxed{; x \leftarrow \widehat{c^*}}$
$b' \xleftarrow{\$} \mathcal{A}^{\mathsf{Decap}_x(\cdot)}(pk, c^*, K_b^*, \boxed{\widehat{c^*}})$
return $[\![b = b']\!]$
```

```
$\mathsf{Decap}_x(c)$:
if $c = x$ then return $\bot$
$K \leftarrow \mathsf{Decap}(sk, c)$
return $K$
```

```
$\mathsf{G}^{\mathsf{SPR\text{-}CCA}}_{\mathsf{K}(,\mathcal{S})}(\mathcal{A})$:
$b \xleftarrow{\$} \{0,1\}$
$(pk, sk, \boxed{\hat{pk}}) \xleftarrow{\$} \mathsf{KGen}()$
$(c_1^*, K_1^*, \boxed{\widehat{c_1^*}}) \xleftarrow{\$} \mathsf{Encap}(pk)$
$c_0^* \xleftarrow{\$} \mathcal{S}$
$\boxed{\widehat{c_0^*} \xleftarrow{\$} \{0,1\}^{\mathsf{cl}} ; c_0^* \leftarrow \mathsf{DecodeCtxt}(\widehat{c_0^*})}$
$K_0^* \xleftarrow{\$} \mathcal{K}$
$x \leftarrow c_b^* \boxed{; x \leftarrow \widehat{c_b^*}}$
$b' \leftarrow \mathcal{A}^{\mathsf{Decap}_x(\cdot)}(pk, c_b^*, K_b^*, \boxed{\widehat{c_b^*}})$
return $[\![b = b']\!]$
```

Figure 2: Security games for IND-CCA **and** SPR-CCA **security of a KEM or obfuscated KEM** K = (KGen, Encap, Decap, . . .) **with key space** $\mathcal{K}$**. Code in** $\boxed{\textbf{dashed boxes}}$ **is only included for an obfuscated KEM; the simulator** $\mathcal{S}$ **is only involved for regular KEMs.**

Definition 2.11 ((Obfuscated) KEM security). *Let* K = (KGen, Encap, Decap, . . .) *be a KEM or obfuscated KEM with key space* $\mathcal{K}$ *and* $\mathsf{G}^{\mathsf{goal\text{-}CCA}}_{\mathsf{K}(,\mathcal{S})}$ *for* goal ∈ {IND, SPR} *be defined as in Figure 2. We say that* K *is* (t, ϵ, q)-goal-CCA*-secure, with respect to a simulator* $\mathcal{S}$ *when* goal = SPR, *if for any adversary* $\mathcal{A}$ *with running time at most* t*, and making at most* q *queries to the* Decap *oracle, we have that*

$$\mathsf{Adv}^{\mathsf{goal\text{-}CCA}}_{\mathsf{K}(,\mathcal{S})}(\mathcal{A}) := 2 \cdot \Pr[\mathsf{G}^{\mathsf{goal\text{-}CCA}}_{\mathsf{K}(,\mathcal{S})}(\mathcal{A}) = 1] - 1 \leq \epsilon.$$

We also use IND-1CCA *security, restricting* $\mathcal{A}$ *to a single* Decap *call.*

We first establish that a sample/encapsulate-and-encode obfuscated KEM maintains IND-CCA security.

Theorem 2.12 (Sample/encapsulate-and-encode obfuscated KEM IND-CCA security). *Let* OKEM *be a sample/encapsulate-and-encode obfuscated KEM based on a regular KEM* KEM *as per Definitions 2.2 and 2.8. For any adversary* $\mathcal{A}$ *against the* IND-CCA *security of* OKEM*, we give an algorithm* $\mathcal{B}$ *such that*

$$\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{OKEM}}(\mathcal{A}) \leq 1/\epsilon^{\mathsf{1succ}}_{\mathsf{OKEM}} \cdot 1/\epsilon^{\mathsf{1encsucc}}_{\mathsf{OKEM}} \cdot \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{KEM}}(\mathcal{B}).$$

Proof. The reduction $\mathcal{B}$ obtains (pk, c^*, K^*) as challenge. It computes the encoded public key as $\hat{pk} \leftarrow \mathsf{Encode}(pk)$ and aborts if encoding fails, which happens with probability at most $\epsilon^{\mathsf{1succ}}_{\mathsf{OKEM}}$. It then computes the encoded ciphertext as $\widehat{c} \xleftarrow{\$} \mathsf{EncodeCtxt}(c^*)$ and again aborts if encoding fails, which happens with probability at most $\epsilon^{\mathsf{1encsucc}}_{\mathsf{OKEM}}$. Run $\mathcal{A}$ on input $(pk, c^*, K^*, \widehat{c^*})$. Upon a $\mathsf{Decap}_x(c)$ query, $\mathcal{B}$ asks DecodeCtxt(c) to its own decapsulation oracle (for unencoded ciphertexts) and returns the response. Finally, $\mathcal{B}$ outputs $\mathcal{A}$'s bit guess as its own. If $\mathcal{B}$ does not abort, it provides a validly distributed simulation for $\mathcal{A}$, which establishes the claim. □

We next show that a sample/encapsulate-and-encode obfuscated KEM maintains SPR-CCA security, if the underlying KEM is SPR-CCA-secure wrt. what we call a "uniform-encapsulation" simulator $\mathcal{S}_{\mathsf{unif}}$ which samples a fresh key pair (sk, pk) and outputs the ciphertext resulting from encapsulating against pk.

Theorem 2.13 (Sample/encapsulate-and-encode obfuscated KEM SPR-CCA security). *Let* OKEM *be a sample/encapsulate-and-encode obfuscated KEM based on a regular KEM* KEM *as per Definitions 2.2 and 2.8. For any adversary* $\mathcal{A}$ *against the* SPR-CCA *security of* OKEM*, we give algorithms* $\mathcal{B}_1$*,* $\mathcal{B}_2$ *such that*

$$\mathsf{Adv}^{\mathsf{SPR\text{-}CCA}}_{\mathsf{OKEM}}(\mathcal{A}) \leq 1/\epsilon^{\mathsf{1succ}}_{\mathsf{OKEM}} \cdot 1/\epsilon^{\mathsf{1encsucc}}_{\mathsf{OKEM}} \cdot \mathsf{Adv}^{\mathsf{SPR\text{-}CCA}}_{\mathsf{KEM},\mathcal{S}_{\mathsf{unif}}}(\mathcal{B}_1)$$
$$+ \mathsf{Adv}^{\mathsf{ctxt\text{-}unif}}_{\mathsf{OKEM}}(\mathcal{B}_2),$$

where $\mathcal{S}_{\mathsf{unif}}$ *is a "uniform-encapsulation" simulator for* KEM.

Proof. We consider an intermediate game G' which behaves like $G^{\text{SPR-CCA}}_{\text{OKEM}}$ for $b = 0$ but omits line 5 and instead keeps $c_0^* \xleftarrow{\$} \mathcal{S}_{\text{unif}}$ sampled by the simulator and computes $\widehat{c_0^*}$ as $\text{EncodeCtxt}(c_0^*)$.

We first bound the difference between $G^{\text{SPR-CCA}}_{\text{OKEM}}$ for $b = 1$ and G' by the SPR-CCA security for simulator $\mathcal{S}_{\text{unif}}$ of the underlying KEM. The reduction $\mathcal{B}_1$ works analogous to the proof of Theorem 2.12, trying to encode both pk and c^* and aborting if this fails, and then simulating the game for $\mathcal{A}$ with these values. (Note that the "uniform-encapsulation" simulator $\mathcal{S}_{\text{unif}}$ (also) follows the distribution in the 1encsucc definition, hence this probability applies to aborting on simulated ciphertexts, too.) Depending on $\mathcal{B}_1$'s challenge bit, this corresponds to either $G^{\text{SPR-CCA}}_{\text{OKEM}}$ for $b = 1$ or G', so we have

$$\Pr[G^{\text{SPR-CCA},b=1}_{\text{OKEM}}] - \Pr[G'] \leq 1/\epsilon^{\text{1succ}}_{\text{OKEM}} \cdot 1/\epsilon^{\text{1encsucc}}_{\text{OKEM}} \cdot \text{Adv}^{\text{SPR-CCA}}_{\text{KEM},\mathcal{S}_{\text{unif}}}(\mathcal{B}_1).$$

We now bound the difference between G' and $G^{\text{SPR-CCA}}_{\text{OKEM}}$ for $b = 0$ by the ciphertext uniformity (ctxt-unif) of OKEM. The reduction $\mathcal{B}_2$ runs KGen itself, obtains $\widehat{c}$ as challenge and sets $c \leftarrow \text{DecodeCtxt}(\widehat{c})$. (Note that $\mathcal{B}_2$ can use the self-generated secret key sk to answer $\mathcal{A}$'s Decap queries.) If $\widehat{c}$ is the real encoded ciphertext, $\mathcal{B}_2$ simulates Game G' encoding a real ciphertext produced by $\mathcal{S}_{\text{unif}}$; else, it simulates $G^{\text{SPR-CCA}}_{\text{OKEM}}$ for $b = 0$. So we have,

$$\Pr[G'] - \Pr[G^{\text{SPR-CCA},b=0}_{\text{OKEM}}] \leq \text{Adv}^{\text{ctxt-unif}}_{\text{OKEM}}(\mathcal{B}_2),$$

which concludes the proof. □

2.4 Obfuscating ML-KEM with Kemeleon

Prior work constructed encodings for Diffie–Hellman public keys to random bytestrings [13, 30, 67], which can be used to construct obfuscated key generation schemes; however, there is not yet any similar construction for public keys used in post-quantum schemes that can be used to construct an obfuscated KEM. In this subsection, we introduce encoding algorithms that work with ML-KEM [46] public keys and ciphertexts, and can be combined with ML-KEM to create an obfuscated KEM. We call our encoding scheme Kemeleon and give its algorithms in Figure 3.[1]

Notation. For a vector $\mathbf{a}$ of length $\text{len}(\mathbf{a})$, $\mathbf{a}[i]$ denotes the ith entry and $\mathbf{a}[i : j]$ the entries between indices i and j, inclusive (where omitting i or j means $i = 1$ resp. $j = \text{len}(\mathbf{a})$). For an integer b, let $b.\text{bit}(i)$ denote the ith bit of the bit representation of b, where $i = 0$ denotes the least significant bit. Additionally, $b.\text{bit}(i : j)$ denotes the bitstring of bits between the ith and jth bit, inclusive. We transparently map between a vector of length k of polynomials of degree n with coefficients in $\mathbb{Z}_q$ and a single vector in $\mathbb{Z}_q^{k \cdot n}$.

Public keys. ML-KEM public keys consist of a vector $\hat{\mathbf{t}}$ of polynomials and a public seed ρ used to generate a public matrix $\hat{\mathbf{A}}$. The public seed ρ is already a string of random bytes, so we focus on the vector of polynomials $\hat{\mathbf{t}}$. Each polynomial in $\hat{\mathbf{t}}$ has coefficients in $\mathbb{Z}_q$ where $q = 3329$. During key generation, each coefficient is stored as a 16-bit integer value. Before the public key is returned, $\hat{\mathbf{t}}$ is serialized into a byte array. Since only 12 of the 16 bits are required to store integers modulo q, every pair of coefficients is mapped to three 8-bit integers during serialization. However, since each coefficient is an integer modulo $q = 3329$, every 12th bit in the output biased toward 0. That is, coefficients do not cover the entire range of integers up to $2^{12} = 4096$, and so the most significant bit of each integer is more likely to be 0 than it is to be 1.

Public Key Encoding/Decoding:

```
Kemeleon.Encode(a):
  $r \leftarrow \sum_{i=1}^{k \cdot n} q^{i-1} \cdot \mathbf{a}[i]$
  if $r.\text{bit}(\lceil \log_2(q^{n \cdot k} + 1) \rceil) = 1$:
      return $\perp$  // most significant bit is 1
  return $r.\text{bit}(0 : \lceil \log_2(q^{n \cdot k} + 1) \rceil - 1)$
```

```
Kemeleon.Decode(r):
  $r.\text{bit}(\lceil \log_2(q^{n \cdot k} + 1) \rceil) \leftarrow 0$
      // set most significant bit to 0
  for $i = 1$ to $k \cdot n$:
      $\mathbf{a}[i] \leftarrow \left( \frac{r - \sum_{j=1}^{i-1} pk[j]}{q^{i-1}} \right) \bmod q$
  return $\mathbf{a}$
```

Ciphertext Encoding/Decoding:

```
Kemeleon.EncodeCtxt(c = (c_1 || c_2)):
  $\mathbf{u} \leftarrow \text{Decompress}_q(c_1, d_u)$
  for $i = 1$ to $k \cdot n$:
      $x \xleftarrow{\$} \left\{ a : \begin{array}{c} \text{Decompress}_q(\text{Compress}_q(.. \\ .. \mathbf{u}[i] + a, d_u), d_u) = \mathbf{u}[i] \end{array} \right\}$
      $\mathbf{u}[i] \leftarrow \mathbf{u}[i] + x$
  $r \leftarrow \text{Kemeleon.Encode}(\mathbf{u})$
  if $r = \perp$ then return $\perp$
  for $i = 1$ to $n$:
      if $c_2[i] = 0$ return $\perp$ with prob. $1/\lceil q/2^{d_v} \rceil$
  return $r \| c_2$
```

```
Kemeleon.DecodeCtxt(r):
  $r_1 \leftarrow r[: \text{len}(r) - d_v \cdot n]$
      // exclude $c_2$
  $r_2 \leftarrow r[d_v \cdot n :]$  // recover $c_2$
  $\mathbf{u} \leftarrow \text{Kemeleon.Decode}(r_1)$
  $c_1 \leftarrow \text{Compress}_q(\mathbf{u}, d_u)$
  $c_2 \leftarrow r_2$
  return $c = (c_1 \| c_2)$
```

Figure 3: Kemeleon **public key and ciphertext encoding and decoding algorithms.**

We use a combination of accumulation and rejection sampling, in line with the sample-and-encode obfuscated key generation paradigm, to map ML-KEM public keys to random strings. The intuition here is to accumulate (sum) the integer coefficients, resulting in a single larger integer whose intermediary bits are no longer biased. Following accumulation, only the single most-significant bit will be biased toward 0. If the most-significant bit in the accumulation is 0, we remove it from the returned string and deterministically set its value to 0 when decoding; if the most-significant bit in the accumulation is 1, we reject and generate a new key pair; this is the rejection sampling step of our algorithm.

The first-sample success probability is the probability that the highest-order bit after accumulation is 0:

$$\epsilon^{\text{1succ}}_{\text{Kemeleon}} = \Pr[r.\text{bit}(\lceil \log_2(q^{n \cdot k} + 1) \rceil) = 0] = \frac{2^{\lceil \log_2(q^{n \cdot k}+1) \rceil - 1}}{q^{n \cdot k}}.$$

This follows from a counting argument: the total number of possible inputs is $q^{n \cdot k}$, and the number of inputs such that the highest-order bit is 0 is the number of integers of bit-length equal to that of $q^{n \cdot k}$ (whose bit-length is $\lceil \log_2(q^{n \cdot k} + 1) \rceil$) where the highest-order bit is zero—this is the number of integers of bit-length $\lceil \log_2(q^{n \cdot k} + 1) \rceil - 1$. For ML-KEM-512 with parameters $q = 3329$, $n = 256$, $k = 2$, this probability is ≈ 0.56, for ML-KEM-768 with $k = 3$ it is ≈ 0.83, and for ML-KEM-1024 with $k = 4$ and $d_v = 5$, it is ≈ 0.62. The advantage of any adversary against public key uniformity is 0 since the output covers the entire output space uniformly.

Ciphertexts. ML-KEM ciphertexts consist of two components, c_1 and c_2, where $c_1 = \text{Compress}_q(\mathbf{u}, d_u)$ is the compression of a vector $\mathbf{u}$ of polynomials and $c_2 = \text{Compress}_q(v, d_v)$ is the compression of a polynomial v (component-wise to d_u and d_v bits,

[1] Our encoding also works for Kyber [7] which has identical public keys and ciphertexts. Naturally, we call that combination Kyber-Kemeleon [18].

Map	Domain	Obfuscation/Ciphertext Uniformity (Defn. 2.4 and 2.10)	First-Sample/Encap Success Probability (Defn. 2.3 and 2.9)		
von Ahn–Hopper [60]	$\mathbb{Z}_p$, s.t. $p = rq + 1$, q is prime, and r coprime to q.	1	$\leq \frac{1}{2}$		
UDH [30]	$\mathbb{Z}_p$, s.t. $p \equiv 3 \bmod 4$, and $q = (p-1)/2$ is prime.	$\approx 2^{-66}$ (MODP 5 & 14)	$\frac{1}{2}$		
Telex [67]	$E(\mathbb{Z}_p) = \{(x, y) : y^2 = x^3 - 3x + b\}$, s.t. $p \equiv 3 \bmod 4$, and E and $E' : -y^2 = x^3 - 3x + b$ have prime order.	$\approx 2^{-160}$ ($p = 2^{168} - 2^8 - 1$)	1		
Elligator2 [13]	$E(GF(q)) = \{(u, v) : v^2 = u^3 + Au^2 + Bu\}$, s.t. $GF(q)$ has odd characteristic and $AB(A^2 - 4B) \neq 0$.	$\approx 2^{-250.83}$ (Curve25519)	$\approx \frac{1}{2}$		
Elligator^2 [58]	$E(GF(q)) = \{(u, v) : v^2 = u^3 + Au^2 + Bu\}$	$\approx 2^{-249.752}$ (Curve25519)	1		
ML-Kemeleon					
— public keys	$(\mathbb{Z}_q[X]/(X^n + 1))^k$	0	≈ 0.56 (ML-KEM-512)	≈ 0.83 (ML-KEM-768)	≈ 0.62 (ML-KEM-1024)
— ciphertexts	$(\mathbb{Z}_{2^{d_u}}[X]/(X^n + 1))^k \times (\mathbb{Z}_{2^{d_v}}[X]/(X^n + 1))$	$\leq 2 \cdot \mathsf{Adv}^{\text{SPR-CCA}}_{\text{ML-KEM},\mathcal{S}_{\$\text{compr}}}(\mathcal{B})$ (cf. Lemma 2.14)	≈ 0.51 (ML-KEM-512)	≈ 0.77 (ML-KEM-768)	≈ 0.57 (ML-KEM-1024)

Table 2: Summary of public key and ciphertext encodings. (UDH/Telex/Elligator^2 interpreted under sample-and-encode paradigm.) Where statistics are given for particular parameters, the parameters or their origin are specified.

respectively). The Compress and Decompress algorithms are defined in [46, Section 4.2.1] as follows, and act component-wise: $\mathsf{Compress}_q(x, d) := \lceil (2^d/q) \cdot x \rfloor \bmod 2^d$ (taking the canonical representative in $[0, 2^d)$) and $\mathsf{Decompress}_q(x, d) := \lceil (q/2^d) \cdot x \rfloor$.

Intuitively, the compression discards some low-order bits of the ciphertext components. As a result of this compression, coefficients in the output ciphertexts are distributed across $\mathbb{Z}_{2^d}$ with some bias due to q not being divisible by 2^d. The compression function, at a high-level, distributes this bias evenly across the space $[0, 2^d)$. Now, there is no longer an overly biased most significant bit (as in the case of public keys); however, the difference in distributions is still significant. Across a vector of k polynomials, each with n coefficients, this further compounds. In particular, for the case of the ML-KEM parameter sets for the c_1 ciphertext, this is non-negligible.

To address the non-uniformity in c_1, we decompress c_1 and then "recover" randomness by adding random values within a designated range. After reintroducing randomness, we have coefficients that are uniformly distributed modulo q, which brings us to the same setting as with public keys. We can then apply the strategy used for public keys on these decompressed and rerandomized values.

While $c_2 = \mathsf{Compress}_q(v, d_v)$, has a similarity non-uniformity, we are able to deal with c_2 in a more space-efficient way due to the specific values of the ML-KEM parameters. For ML-KEM, we have $d_v \in \{4, 5\}$, so $q \bmod 2^{d_v} = 1$, and hence the distribution of compressed ciphertexts is nearly uniform. Only the value 0 occurs with slightly higher probability than other outputs (increased by $1/\lceil q/2^{d_v} \rceil$). In this case, we can perform rejection sampling: for each coefficient of c_2 equal to 0, reject (output $\perp$) with probability $1/\lceil q/2^{d_v} \rceil$. This results in an exactly uniform distribution. Since c_2 is a single polynomial with $n = 256$ coefficients, the probability that a given c_2 is successfully encoded is $(1 - 1/(2^{d_v} \cdot \lceil q/2^{d_v} \rceil))^n$, which is ≈ 0.926 for $d_v = 4$ and ≈ 0.927 for $d_v = 5$. Note that applying this strategy to the first ML-KEM ciphertext element c_1 would result in an unacceptably small success probability, which is why we opt for the decompression and randomness recovery strategy for c_1.

Tallying the results for c_1 and c_2, the statistical distance between uniformly random bit strings and Kemeleon-encoded ML-KEM ciphertexts which are the compression of uniformly random values $(\mathbf{u}, v) \xleftarrow{\$} \mathbb{Z}_q^{k \cdot n} \times \mathbb{Z}_q^n$ is 0. We note that this can easily be adapted to byte strings via random padding to a multiple of 8 bits. The first-encap success probability (Definition 2.9) is the probability that $\mathsf{Encode}(u) \neq \perp$ in line 5 of EncodeCtxt and that the second element is not rejected:

$$\epsilon^{\text{1encsucc}}_{\text{Kemeleon}} = \frac{2^{\lceil \log_2(q^{n \cdot k}+1) \rceil - 1}}{q^{n \cdot k}} \cdot \left(1 - \frac{1}{2^{d_v} \cdot \lceil q/2^{d_v} \rceil}\right)^n .$$

For ML-KEM-512 with parameters $q = 3329$, $n = 256$, $k = 2$, $d_v = 4$, this probability is ≈ 0.51, for ML-KEM-768 with $k = 3$, $d_v = 4$, it is ≈ 0.77, and for ML-KEM-1024 with $k = 4$, $d_v = 5$, it is ≈ 0.57.

ML-Kemeleon. We can now instantiate an obfuscated KEM using Kemeleon as the obfuscator. In particular, we define ML-Kemeleon to be the sample/encapsulate-and-encode obfuscated KEM (Definition 2.8) based on ML-KEM and using the Kemeleon public key and ciphertext encoding/decoding algorithms from Figure 3.

We establish the following bound for the ciphertext uniformity of ML-Kemeleon.

Lemma 2.14 (ctxt-unif of ML-Kemeleon). *For any adversary $\mathcal{A}$ against the ciphertext uniformity (Definition 2.10) of* ML-Kemeleon, *we give an algorithm $\mathcal{B}$ wrt. a simulator $\mathcal{S}_{\$\text{compr}}$ that samples uniformly random ciphertexts $(\mathbf{u}, v)$ before compression and outputs $c = \mathsf{Compress}_q(\mathbf{u}, d_u) \| \mathsf{Compress}_q(v, d_v)$, such that*

$$\mathsf{Adv}^{\text{ctxt-unif}}_{\text{ML-Kemeleon}}(\mathcal{A}) \leq 1/\epsilon^{\text{1encsucc}}_{\text{Kemeleon}} \cdot \mathsf{Adv}^{\text{SPR-CCA}}_{\text{ML-KEM},\mathcal{S}_{\$\text{compr}}}(\mathcal{B}).$$

Proof. Denote by G_0 the $b = 1$ version of the ctxt-unif challenge (sampling and encoding a freshly encapsulated ciphertext) and by

G_1 the $b = 0$ version (sampling the challenge uniformly from $\{0,1\}^{\mathsf{cl}}$), i.e., $\mathsf{Adv}^{\mathsf{ctxt\text{-}unif}}_{\mathsf{ML\text{-}Kemeleon}}(\mathcal{A}) = \Pr[G_1] - \Pr[G_0]$ by standard advantage rewriting. We consider an intermediate game G' which runs obtains a ciphertext $c \xleftarrow{\$} \mathcal{S}_{\$\mathsf{compr}}$ from the simulator and encodes it into $\widehat{c} \leftarrow \mathsf{Kemeleon.EncodeCtxt}(c)$.

We first bound the difference between G_1 and G' by the SPR-CCA security of ML-KEM for simulator $\mathcal{S}_{\$\mathsf{compr}}$. The reduction obtains (pk, c^*, K^*) and encodes $\widehat{c^*} \leftarrow \mathsf{Kemeleon.EncodeCtxt}(c^*)$, aborting if the latter fails (which happens with probability bounded by $\epsilon^{\mathsf{1encsucc}}_{\mathsf{ML\text{-}Kemeleon}}$). It then runs $\mathcal{A}$ on input c^* and $\mathcal{A}$'s bit guess as its own. Depending on $\mathcal{B}_1$'s challenge bit, this corresponds to either G_1 or G', so we have

$$\Pr[G_1] - \Pr[G'] \leq 1/\epsilon^{\mathsf{1encsucc}}_{\mathsf{ML\text{-}Kemeleon}} \cdot \mathsf{Adv}^{\mathsf{SPR\text{-}CCA}}_{\mathsf{ML\text{-}KEM},\mathcal{S}_{\$\mathsf{compr}}}(\mathcal{B}).$$

Now, between G' and G_0, $\mathcal{A}$ must distinguish between the encoding of uniformly random ciphertexts $(\mathbf{u}, v)$ before compression, as output by $\mathcal{S}_{\$\mathsf{compr}}$, and uniformly random bit strings from $\{0,1\}^{\mathsf{cl}}$. But as previously established, the statistical distance between these distributions is 0. So, we have that $\Pr[G'] - \Pr[G_0] = 0$ and the theorem statement follows. □

The properties of public key and ciphertext mappings are summarized in Table 2, and the analysis of UDH and Telex is deferred to Appendix B.

3 The `obfs4/lyrebird` Protocol

The `obfs4/lyrebird` protocol, specified in [56][2], is separated into two distinct phases: a *key exchange* phase and a *data transfer* phase. We refer to the protocol as `obfs4`, but note that the implementation on which we base our analysis has been renamed to `lyrebird`. We are interested specifically in the key exchange protocol, which is shown in Figure 4. The `obfs4` key exchange protocol consists of two messages exchanged between the client and the server.

Client to server message. The client begins by performing a sample-and-encode obfuscated key generation, in which it samples an emphemeral X25519 [40] Diffie–Hellman secret x and computes corresponding public key X and the Elligator2-encoded [13] obfuscated public key X'. The client generates padding, P_C, of random length and then computes a MAC tag, M_C, keyed with semi-secret bridge information (a DH public key, B, and a 160-bit string, *NodeID*), of the obfuscated public key X'. Finally, it computes a MAC tag, MAC_C, over the three previous components of the message: X', P_C, and M_C. It sends the obfuscated public key together with the random padding and the two MAC tags. Here, the random padding is intended to vary the length of initial messages, contributing to obfuscation properties. The final MAC tag authenticates the message and the intermediate MAC tag is intended to prevent clients who do not know the bridge information (B and *NodeID*) from connecting to the server.

We note that the intermediate tag M_C is not necessary, neither for parsing the message nor for preventing active probing, since the bridge information also acts as a key in MAC_C and a server can use the known length of the obfuscated public key and the final MAC tag to parse the message from the end.

Server to client message. Upon receipt of the client message, the server computes M_C and verifies that it appears as expected in the message. It then verifies the final MAC tag MAC_C. If either verification fails or MAC_C has already been seen by the server, then the server aborts and does not respond to the client. Otherwise, it continues with the protocol, decoding the obfuscated public key and generating its own Elligator2-encoded, obfuscated X25519 ephemeral keys, y, Y, Y'. It then performs the server-side computation of an `ntor` [31] handshake using its ephemeral (y, Y) and long-term (b, B) keys, and the client's public key X. Similar to the client, the server generates random padding P_S and computes two MAC tags, M_S and MAC_S. Here, M_S is necessary for the client to be able to parse the message, because the server may continue sending data following the final authentication tag, preventing the client from parsing the message from the end. The server returns to the client its obfuscated public key Y', the `ntor` authentication tag *auth*, padding P_S, and the two MAC tags M_S and MAC_S.

Upon receipt, the client similarly verifies both MAC values and then performs the remainder of the client-side operations of the `ntor` handshake. The `ntor` component includes the computation of the session key, *skey*, and enables authentication of the server.

Epochs. To manage the state overhead for replay protection, `obfs4` works in epochs. The client signals its epoch and includes it under the value MAC_C. The server then checks that this epoch is off by at most one from its own epoch. This allows the server to store the set of MAC values seen in its state $\mathcal{S}_{MAC}$ *per epoch*, and clear that storage for an epoch once the after next one is reached. To simplify the presentation, we omit epochs here.

4 Obfuscated Key Exchange

4.1 Security Goals

An obfuscated key exchange protocol aims to satisfy traditional key exchange properties and those specific to the setting of censorship circumvention. Some of our goals are motivated by properties that are used in practice by censors to classify obfuscated traffic (discussed in Section 8.2). Our goals are also motivated by the explicit threat models of FEPs, such as those in [56, Section 2], that have developed over time in reaction to increased capabilities of censors. First we explain the security goals of our model, then formalize security using a Bellare–Rogaway-style key exchange model [12].

(1) **Key indistinguishability.** Secrecy of the shared key established in the protocol is through indistinguishability from random, as in Bellare–Rogaway [12]. This is not related to any explicit `obfs4` goal, but is a clear goal of the key exchange protocol. The aim of the adversary is to guess the challenge bit b used in Test queries which return a real or random session key. Any tested session must be *fresh*: the session key cannot have been revealed (on a session or its partnered session). Further, we consider key indistinguishability against passive executions (sessions have honest partners) and with *forward secrecy*, i.e., the responder's long-term secret key must be unrevealed prior to acceptance. The adversary wins if it correctly guesses the challenge bit without violating freshness.

(2) **Obfuscation.** Intuitively the obfuscation property should capture a protocol's ability to be unidentifiable or fingerprintable by an adversary. From the `obfs4` threat model [56, §2]:

[2] We base our analysis on the implementation from The Tor Project [49].

```
Server key generation/setup
$NodeID \xleftarrow{\$} \{0,1\}^{160}$
$(b, B) \xleftarrow{\$} \mathsf{KGen}_{\mathsf{X25519}}()$
$st.\mathcal{S}_{MAC} \leftarrow \emptyset$
return $((b, NodeID), (B, NodeID), st)$

Client   knows $(B, NodeID)$                                knows $(b, NodeID)$   Server
$(x, X, X') \xleftarrow{\$} \mathsf{X2Ell2.KGen}()$
$P_C \xleftarrow{\$} \{0,1\}^{\mathsf{pmin_C..pmax_C}}$
$M_C \leftarrow \mathsf{HMAC}(B \| NodeID, X')$
$MAC_C \leftarrow \mathsf{HMAC}(B \| NodeID, X' \| P_C \| M_C)$

            $\xrightarrow{msg_C = X' \| P_C \| M_C \| MAC_C}$

        $X' \leftarrow msg_C[1..\mathrm{ol}]$
        $M_C \leftarrow \mathsf{HMAC}(B \| NodeID, X')$
        parse $(X' \| P_C \| M_C \| MAC_C) \leftarrow msg_C$ using $M_C$; else break
        if $\mathsf{HMAC}(B \| NodeID, X' \| P_C \| M_C) \neq MAC_C$: break
        if $MAC_C \in st.\mathcal{S}_{MAC}$: break
        $st.\mathcal{S}_{MAC} \leftarrow st.\mathcal{S}_{MAC} \cup \{MAC_C\}$
        $X \leftarrow \mathsf{X2Ell2.Decode}(X')$
        $(y, Y, Y') \xleftarrow{\$} \mathsf{X2Ell2.KGen}()$
        ntor handshake component
            $protoID \leftarrow$ “ntor-curve25519-sha256-1”
            $secret\_input \leftarrow X^y \| X^b \| B \| B \| X \| Y \| protoID \| NodeID$
            $skey \leftarrow \mathsf{HMAC}(secret\_input,$ “:key_extract”$)$
            $verify \leftarrow \mathsf{HMAC}(secret\_input,$ “:key_verify”$)$
            $auth\_input \leftarrow verify \| B \| B \| X \| Y \| protoID \| NodeID \|$“Server”
            $auth \leftarrow \mathsf{HMAC}(auth\_input,$ “:mac”$)$
        $P_S \xleftarrow{\$} \{0,1\}^{\mathsf{pmin_S..pmax_S}}$
        $M_S \leftarrow \mathsf{HMAC}(B \| NodeID, Y')$
        $MAC_S \leftarrow \mathsf{HMAC}(B \| NodeID, Y' \| auth \| P_S \| M_S)$

            $\xleftarrow{msg_S = Y' \| auth \| P_S \| M_S \| MAC_S}$

$Y' \leftarrow msg_S[1..\mathrm{ol}]$
$M_S \leftarrow \mathsf{HMAC}(B \| NodeID, Y')$
parse $(Y' \| auth \| P_S \| M_S \| MAC_S) \leftarrow msg_S$ using $M_S$; else break
if $\mathsf{HMAC}(B \| NodeID, Y' \| auth \| P_S \| M_S) \neq MAC_S$: break
$Y \leftarrow \mathsf{X2Ell2.Decode}(Y')$
ntor handshake component
    $protoID \leftarrow$ “ntor-curve25519-sha256-1”
    $secret\_input \leftarrow Y^x \| B^x \| B \| B \| X \| Y \| protoID \| NodeID$
    $skey \leftarrow \mathsf{HMAC}(secret\_input,$ “:key_extract”$)$
    $verify \leftarrow \mathsf{HMAC}(secret\_input,$ “:key_verify”$)$
    $auth\_input \leftarrow verify \| B \| B \| X \| Y \| protoID \| NodeID \|$“Server”
    if $\mathsf{HMAC}(auth\_input,$ “:mac”$) \neq auth$: break
```

Figure 4: The `obfs4` [56] obfuscated key exchange protocol.

> “obfs4 offers protection against passive Deep Packet Inspection machines that expect the obfs4 protocol. Such machines should not be able to verify the existence of the obfs4 protocol without obtaining the server’s Node ID and identity public key. [...] obfs4 offers protection against some non-content protocol fingerprints, specifically the packet size, and optionally packet timing.”

We observe that protocols can be divided into *classes*, where a class of protocols determines its traffic pattern, message sequence, etc. (e.g., all TLS traffic may lie in one class while all Skype traffic lies in another). Obfuscation is defined with respect to a simulator $\mathcal{S}$ that defines a class of protocols. Then, real executions should look indistinguishable from simulated messages exchanged. Ideally, the simulator captures a class that hides a protocol from identification.

Our simulator-based definition is generic and adaptable, and we discuss properties for “good” simulators in Section 8.2. We prove obfuscation of our protocols with respect to a simulator that outputs all-random messages of varying lengths. The simulator-based approach enables us to capture both types of circumvention protocols defined in [59]: *polymorphic*-style protocols (which aim to hide identifying features to evade blacklisting), and *steganographic*-style protocols (which aim to appear like a white-listed protocol). Obfuscation is modeled via a ChallExec oracle and ObfFresh predicate. The aim of the adversary is to guess the challenge bit b used in ChallExec queries which return either the real transcript and key from a protocol run, or a simulated transcript and random key.[3] The ObfFresh predicate captures two variants of obfuscation: *regular* obfuscation, which requires that the server’s *public* key is not revealed, and *strong* obfuscation, which requires that the server’s *secret* key is not revealed.

(3) **Probing resistance.** Following evidence of censors actively probing suspicious proxy servers [23] and evidence of the efficacy of probing against FEPs [28], the following requirement was included in the `obfs4` threat model [56, §2]:

> “obfs4 offers protection against active attackers attempting to probe for obfs4 servers. Such machines should not be able to verify the existence of an obfs4 server without obtaining the server’s Node ID and identity public key.”

Servers should not respond to active probing attacks, in which the client does not know the server’s public key. This is modeled within the normal Bellare–Rogaway-style Send oracle via a Probed flag. The model tracks the first messages sent by initiators in a list $\mathcal{F}$. Once a message is delivered to a responder, it is removed from $\mathcal{F}$. If the adversary successfully elicits a non-empty response from a responder, whose public key is not revealed, in response to a message not in $\mathcal{F}$, then the adversary wins. The list $\mathcal{F}$ ensures that an adversary cannot trivially win by forwarding the first message from an honest initiator to a responder and receiving a response.

(4) **Explicit authentication.** Finally, our model ensures explicit authentication of the server. An adversary wins if it causes a client to accept a session without a partnered server session existing. This property is modeled via the ExplicitAuth predicate and captures the following statement from the `obfs4` threat model [56, §2]:

> “obfs4 offers protection against active attackers that have obtained the server’s Node ID and identity public key. Such machines should not be able to impersonate the server without obtaining the server’s identity private key.”

4.2 Key Exchange and Model Syntax

We specify a key exchange protocol KE through two algorithms:

- Setup$(id, role) \xrightarrow{\$} (sk, pk, st)$ generates the public-secret key pair (pk, sk) as well as the initial user state st for a protocol user with identity id in role $role \in \{\mathsf{initiator}, \mathsf{responder}\}$.
- Run$(\pi_u^i, st_u, sk_u, pk_v, m) \xrightarrow{\$} (\pi_u^i, st_u, m')$ processes a protocol message m delivered to session π_u^i following the protocol specification (along with inputs the session owner’s state st_u and

[3] The steganographic key exchange security notion from [60] can be seen as a single ChallExec oracle run (one real-or-random trace, with real-or-random key).

secret key sk_u, and the session's peer public key pk_v), updates π_u^i and st_u accordingly, and outputs the response message m'.

Further, we write $\mathsf{KE.KS}$ to denote the session key space of KE.

4.2.1 Session and game variables. Session object π_u^i captures the session information for the ith session owned by user u. Each user u is assigned a role, $u.role \in \{\mathsf{initiator}, \mathsf{responder}\}$, and acts accordingly as initiator or responder in the protocol.

Each session object π_u^i holds several variables. The following session variables in *italics* font are accessible by the key exchange protocol (i.e., Run):

- $\pi_u^i.peerid$: the identity of the session's intended peer.
- $\pi_u^i.status$: the state of execution (initially running, then set once by the protocol to accepted or rejected).
- $\pi_u^i.skey$: the session key.
- $\pi_u^i.sid$, $\pi_u^i.cid$: the session and contributive identifiers.

These folllowing session variables in sans-serif font are accessible by the security game only:

- $\pi_u^i.\mathsf{t_{acc}}$: the time at which the session accepted (initially ∞).
- $\pi_u^i.\mathsf{revealed}$, $\pi_u^i.\mathsf{tested}$: flags indicating whether the session was revealed or tested, respectively.

The security game further tracks the following game variables:

- time: a logical clock to order queries by the adversary.
- users: the number of users in the game.
- b: the challenge bit.
- sk_u, pk_u: the secret and public key of user u.
- revsk_u, revpk_u: flags indicating whether the secret or public, key of user u was revealed.

For syntactical convenience, we will interpret variables which are unset or set to ∞ as false in boolean conditions.

4.2.2 Session identifiers, contributive identifiers, and partnering. We use *session identifiers* [11] to determine that two sessions π_u^i, π_v^j are *partnered* if and only if $\pi_u^i.sid = \pi_v^j.sid$. Partnering is used to define basic correctness/soundness properties and to exclude trivial attacks like testing and revealing two partnered sessions that jointly executed the protocol.

We further use *contributive identifiers* [20] to determine when a responder session has an honest communication partner. Recall that initiators are unauthenticated. Hence, to avoid trivial attacks, the adversary may test a responder session only if that session honestly received the values specified in the contributive identifier *cid*.

4.2.3 Security Definition

Definition 4.1 (Obfuscated key exchange security)**.** *Let* KE *be a key exchange protocol and* $\mathsf{G}^{\mathsf{ObfKE}}_{\mathsf{KE},\mathcal{S}}(\mathcal{A})$ *be the obfuscated key exchange security game wrt. a simulator* $\mathcal{S}$ *defined in Figure 5 for an adversary* $\mathcal{A}$*. We define the advantage of* $\mathcal{A}$ *in breaking the* ObfKE *security of* KE *as*

$$\mathsf{Adv}^{\mathsf{ObfKE}}_{\mathsf{KE},\mathcal{S}}(\mathcal{A}) := 2 \cdot \Pr\left[\mathsf{G}^{\mathsf{ObfKE}}_{\mathsf{KE},\mathcal{S}}(\mathcal{A}) \Rightarrow 1\right] - 1.$$

We distinguish two flavors of ObfKE*, capturing regular (*rObfKE*) and strong (*sObfKE*) obfuscation through different* ObfFresh *predicates (Figure 5); we omit the prefixes if the flavor is clear from context.*

4.2.4 Single-challenge selective security For our security analyses, it will be convenient to establish security in a simpler version of the obfuscated key exchange game, where the adversary has to commit to winning via either:

(1) a single TEST query to some pre-declared session, or
(2) a single (or, optionally, multiple) CHALLEXEC queries against some pre-declared server.

We denote these variants as ObfKE-1 (for a single CHALLEXEC query) and ObfKE-1* (for multiple CHALLEXEC queries). These are weaker versions of the main game, asking only for selective security in a single-challenge setting. Yet we show in the following that these simpler versions generically imply full security (per Definition 4.1) via a hybrid argument, for the same type of obfuscation (regular or strong). The hybrid works via guessing which session or server the adversary will challenge, losing a factor $(n_s + n_r q_{\mathrm{C}})$ when reducing ObfKE to ObfKE-1 security, where n_s and n_r are the number of sessions and servers in the game and q_{C} the number of CHALLEXEC queries the adversary makes. For ObfKE-1*, allowing multiple CHALLEXEC queries to the pre-declared server, the loss is only $(n_s + n_r)$.

Definition 4.2 (Single-challenge selective obfuscated key exchange security)**.** *Let* KE *be a key exchange protocol and* $\mathsf{G}^{\mathsf{ObfKE\text{-}1}}_{\mathsf{KE},\mathcal{S}}(\mathcal{A})$ *be the obfuscated key exchange security game wrt. a simulator* $\mathcal{S}$ *defined in Figure 5 for an adversary* $\mathcal{A}$*, with the following restrictions:*

(1) At the outset of the game (as input to INITIALIZE*),* $\mathcal{A}$ *has to commit on winning via a* TEST *query or* CHALLEXEC *queries by outputting an attack type* A*, where* $\mathrm{A} \in \{$TEST, CHALLEXEC$\}$.*
(2) If $\mathrm{A} =$ TEST*,* $\mathcal{A}$ *has to further commit to a session index* $s \in [1..n_s]$ *at the outset of the game. In the game,* $\mathcal{A}$ *can only make a single* TEST *query which has to be on the* s*th created session. The game penalizes* $\mathcal{A}$ *by setting* $b' \leftarrow 0$ *if it makes any other* TEST *query, or any query to* CHALLEXEC.*
(3) If $\mathrm{A} =$ CHALLEXEC*,* $\mathcal{A}$ *has to further commit to a user index* $p \in [1..n_r]$ *at the outset of the game. In the game,* $\mathcal{A}$ *can only make a single* CHALLEXEC(u, v) *query with* v *being the* p*th created server. The game penalizes* $\mathcal{A}$ *by setting* $b' \leftarrow 0$ *if it makes any other* CHALLEXEC *query, or any query to* TEST.*

We define the advantage of $\mathcal{A}$ *in breaking the ("single-challenge selective")* ObfKE-1 *security of* KE *as*

$$\mathsf{Adv}^{\mathsf{ObfKE\text{-}1}}_{\mathsf{KE},\mathcal{S}}(\mathcal{A}) := 2 \cdot \Pr\left[\mathsf{G}^{\mathsf{ObfKE\text{-}1}}_{\mathsf{KE},\mathcal{S}}(\mathcal{A}) \Rightarrow 1\right] - 1.$$

We also consider a ObfKE-1* *variant* $\mathsf{G}^{\mathsf{ObfKE\text{-}1^*}}$ *where for* $\mathrm{A} =$ CHALLEXEC*,* $\mathcal{A}$ *is allowed to make* multiple CHALLEXEC(u, v) *queries with* v *being the* p*th created server.*

*As for the full security game (cf. Definition 4.1), we distinguish regular (*rObfKE-1, rObfKE-1**) and strong (*sObfKE-1, sObfKE-1**) obfuscation through different* ObfFresh *predicates (Figure 5); we omit the prefixes if the flavor is clear from context.*

Clearly, ObfKE security implies ObfKE-1 (and ObfKE-1*) security, since a successful adversary against the latter is also one against the former where no commitment or query restrictions apply. We next show that the reverse also holds via a hybrid argument.

```
$\underline{G^{\text{ObfKE}}_{\text{KE},\mathcal{S}}(\mathcal{A})}$

INITIALIZE:
$\text{time} \leftarrow 0; \text{users} \leftarrow 0$
$\text{b} \xleftarrow{\$} \{0, 1\}$
$\text{Probed} \leftarrow \text{false}$

NEWUSER$(id, role)$:
$u \leftarrow \text{++users}$
$u.role \leftarrow role$
$(pk_u, sk_u, st_u) \xleftarrow{\$} \text{Setup}(id, role)$
$\text{revsk}_u \leftarrow \infty\ ; \text{revpk}_u \leftarrow \infty$
```

```
REVSESSIONKEY$(u, i)$:
if $\pi^i_u = \bot$ or $\neg\pi^i_u.\text{t}_{\text{acc}}$ then return $\bot$
$\pi^i_u$.revealed $\leftarrow$ true
return $\pi^i_u.skey$

REVSECRETKEY$(u)$:
if $\text{revsk}_u$ then return $\bot$
$\text{revsk}_u \leftarrow$ ++time
if $\neg\text{revpk}_u$ then $\text{revpk}_u \leftarrow$ time // Consider $pk_u$ revealed, too
return $sk_u$

REVPUBLICKEY$(u)$:
if $\text{revpk}_u$ then return $\bot$
$\text{revpk}_u \leftarrow$ ++time
return $pk_u$
```

```
TEST$(u, i)$:
if $\pi^i_u = \bot$ or $\neg\pi^i_u.\text{t}_{\text{acc}}$ or $\pi^i_u$.tested then
    return $\bot$
$\pi^i_u$.tested $\leftarrow$ true
$k_1 \leftarrow \pi^i_u.skey$
$k_0 \xleftarrow{\$} \text{KE.KS}$
return $k_\text{b}$

FINALIZE$(\text{b}')$:
if $\neg$Sound then $\text{b}' \leftarrow \text{b}$
if $\neg$ExplicitAuth then $\text{b}' \leftarrow \text{b}$
if Probed then $\text{b}' \leftarrow \text{b}$
if $\neg$Fresh then $\text{b}' \leftarrow 0$
if $\neg$ObfFresh then $\text{b}' \leftarrow 0$
return $[\![\text{b} = \text{b}']\!]$
```

```
SEND$(u, i, m)$:
$f \leftarrow [\![\pi^i_u = \bot]\!]$ // First SEND to this session?
if $\pi^i_u = \bot$ and $u.role$ = initiator then
    $\pi^i_u.peerid \leftarrow m; m \leftarrow \varepsilon$ // "Virtual" first message to initator sets peer id
if $\pi^i_u.status \in \{\text{accepted, rejected}\}$ then return $\bot$
$(\pi^i_u, st_u, m') \xleftarrow{\$} \text{Run}(\pi^i_u, st_u, sk_u, pk_{\pi^i_u.peerid}, m)$
if $\pi^i_u.status$ = accepted then $\pi^i_u.\text{t}_{\text{acc}} \leftarrow$ ++time
if $f$ then // Upon first message sent/received...
    if $u.role$ = initiator then
        $\mathcal{F}_{\pi^i_u.peerid} \leftarrow \mathcal{F}_{\pi^i_u.peerid} \cup \{m'\}$ // Record initators' first messages (per responder)
    if $u.role$ = responder and $m' \neq \varepsilon$ then
        if $m \notin \mathcal{F}_u$ and $\neg\text{revpk}_u$ then Probed $\leftarrow$ true // Response to a non–initiator-first message by a non–$pk$-revelead responder
is a successful probe
        $\mathcal{F}_u \leftarrow \mathcal{F}_u \setminus \{m\}$ // Consider $m$ "consumed"
return $(\pi^i_u.status, m')$
```

```
CHALLEXEC$(u, v)$:
$\pi_u \leftarrow \pi_v \leftarrow \bot$ // Temporary initator and responder sessions
if $u.role \neq$ initiator or $v.role \neq$ responder then return $\bot$
$trans_1 \leftarrow (); \pi_u.peerid \leftarrow v; p \leftarrow u; m \leftarrow \varepsilon$
repeat // Execute entire protocol and collect real transcript
    $(\pi_p, st_p, m) \xleftarrow{\$} \text{Run}(\pi_p, st_p, sk_p, pk_{\pi_p.peerid}, m)$
    $trans_1 \leftarrow trans_1 \| (m)$
    if $p = u$ then $p \leftarrow v$ else $p \leftarrow u$ // Switch parties
until $\pi_u.status = \pi_v.status$ = accepted
$trans_0 \xleftarrow{\$} \mathcal{S}()$ // Simulated transcript
$k_1 \leftarrow \pi_u.skey; k_0 \xleftarrow{\$} \text{KE.KS}$ // Real-or-random session key
$\text{chall}_v \leftarrow$ true // Mark server $v$ as challenged
return $(trans_\text{b}, k_\text{b})$
```

```
Fresh:
for each $\pi^i_u : \pi^i_u$.tested
    if $\pi^i_u$.revealed then
        return false // tested session may not be revealed
    if $\exists\pi^j_v \neq \pi^i_u : \pi^j_v.sid = \pi^i_u.sid \wedge (\pi^j_v.\text{tested} \vee \pi^j_v.\text{revealed})$ then
        return false // tested session's partnered session may not be tested or revealed
    if $u.role = \text{initiator} \wedge \text{revsk}_{\pi^i_u.peerid} \leq \pi^i_u.\text{t}_{\text{acc}} \wedge \neg\exists\pi^j_v \neq \pi^i_u : (\pi^i_u.sid = \pi^j_v.sid)$
        return false // initiators: forward secrecy (peer's $sk$ unrevealed prior to acceptance) or passive execution
    if $u.role = \text{responder} \wedge \neg\exists\pi^j_v \neq \pi^i_u : (\pi^i_u.cid = \pi^j_v.cid \wedge v.role = \text{initiator})$
        return false // responders: security only for passive executions (as initiators are unauthenticated)
return true

ObfFresh:
$\boxed{\text{if } \exists v : \text{revpk}_v \wedge \text{chall}_v \text{ then}}$ // Regular obfuscation (rObfKE)
if $\exists v : \text{revsk}_v \wedge \text{chall}_v$ then // Strong obfuscation (sObfKE)
    return false // Challenge pk revealed (fwd-secret: prior to challenge)
return true
```

```
ExplicitAuth:
    // Explicit authentication of responders to initiators
$\forall\pi^i_u$: ( $u.role = \text{initiator} \wedge \pi^i_u.\text{t}_{\text{acc}} < \text{revsk}_{\pi^i_u.peerid}$
            $\implies \exists\pi^j_v : v = \pi^i_u.peerid \wedge \pi^i_u.sid = \pi^j_v.sid$ )

Sound:
if $\exists$ distinct $\pi^i_u, \pi^j_v, \pi^k_w : \pi^i_u.sid = \pi^j_v.sid = \pi^k_w.sid \neq \bot$ then
    return false // no triple sid match
if $\exists\pi^i_u, \pi^j_v : \pi^i_u.sid = \pi^j_v.sid \neq \bot \wedge u.role = v.role$ then
    return false // partnering implies different roles
if $\exists\pi^i_u, \pi^j_v : \pi^i_u.sid = \pi^j_v.sid \neq \bot \wedge \pi^i_u.cid \neq \pi^j_v.cid$ then
    return false // partnering implies same contributive identifiers
if $\exists\pi^i_u, \pi^j_v : \pi^i_u.sid = \pi^j_v.sid \neq \bot \wedge u.role = \text{initiator} \wedge \pi^i_u.peerid \neq v$ then
    return false // partnering implies agreement on responder ID
if $\exists\pi^i_u, \pi^j_v : \pi^i_u.sid = \pi^j_v.sid \neq \bot \wedge \pi^i_u.skey \neq \pi^j_v.skey$ then
    return false // partnering implies same key
return true
```

Figure 5: Obfuscated key exchange security (ObfKE) **game (top) capturing key indistinguishability, explicit server authentication, (** $\boxed{\textbf{regular (rObfKE)}}$ **or strong (sObfKE)) obfuscation with respect to a simulator** $\mathcal{S}$**, and probing resistance, via predicates (bottom)** Fresh, ExplicitAuth, ObfFresh, **and flag** Probed, **respectively.**

Theorem 4.3 (ObfKE-1, ObfKE-1* $\implies$ ObfKE). *Let* KE *be a key exchange protocol. Then for any adversary $\mathcal{A}$ against the* ObfKE *security of* KE *interacting with at most n_s sessions and n_r servers, there is an adversary $\mathcal{B}$ against the* ObfKE-1 *or* ObfKE-1* *security of* KE *such that*

$$\text{Adv}^{\text{ObfKE}}_{\text{KE},\mathcal{S}}(\mathcal{A}) \leq (n_s + n_r \cdot q_C) \cdot \text{Adv}^{\text{ObfKE-1}}_{\text{KE},\mathcal{S}}(\mathcal{B}),$$
$$\text{Adv}^{\text{ObfKE}}_{\text{KE},\mathcal{S}}(\mathcal{A}) \leq (n_s + n_r) \cdot \text{Adv}^{\text{ObfKE-1}^*}_{\text{KE},\mathcal{S}}(\mathcal{B}).$$

The relations hold for both regular and strong obfuscation flavors of the respective games.

We defer the proofs for the ObfKE-1 and ObfKE-1* hybrid results in Theorem 4.3 to Appendix C.

5 Security Analysis of `obfs4`

We now analyze the security of `obfs4` in our model, showing that it achieves rObfKE security (i.e., with regular obfuscation). We rely on the gap Diffie–Hellman (GapDH) problem [47] being hard for X25519, modeling HMAC as a random oracle, and the public key uniformity of sample-and-encode obfuscated X25519 [40] DH shares with Elligator2 encoding [13] (which we computed in Section 2.1).

Simulator definition. We establish rObfKE security with respect to the following simulator $\mathcal{S}_{\mathtt{obfs4}}$ which outputs two uniformly random messages of length corresponding to the `obfs4` message elements including random-length padding:

```
$\underline{\mathcal{S}_{\mathtt{obfs4}}:}$
$r_C \xleftarrow{\$} [\mathrm{pmin_C}, \mathrm{pmax_C}]$ ; $r_S \xleftarrow{\$} [\mathrm{pmin_S}, \mathrm{pmax_S}]$   // random-length client/server padding
$m_1 \xleftarrow{\$} \{0,1\}^{\mathrm{ol}+512+r_C}$   // 1x encoded DH + 2x HMAC outputs + client padding
$m_2 \xleftarrow{\$} \{0,1\}^{\mathrm{ol}+768+r_S}$   // 1x encoded DH + 3x HMAC outputs + server padding
return $(m_1, m_2)$
```

Session and contributive identifiers. We set the session identifier

$$sid := (X, Y, B, NodeID),$$

where X, Y are the sent/received initiator/responder DH shares, and $(B, NodeID)$ is the responder's public key. Sessions set the contributive identifier to

$$cid := (X)$$

upon the client sending or the server receiving the first message.

We now give the theorem statement and a proof sketch; the complete proof is deferred to Appendix D.

Theorem 5.1. *Let* `obfs4` *be defined as in Figure 4. Assume the* GapDH *problem is hard for* X25519 *and that* HMAC *behaves like a random oracle. For any* rObfKE *adversary* $\mathcal{A}$ *against* `obfs4`*, we give algorithms* $\mathcal{B}_1$*,* $\mathcal{B}_2$*,* $\mathcal{B}_3$ *in the proof such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{rObfKE}}_{\mathtt{obfs4},\mathcal{S}_{\mathtt{obfs4}}}(\mathcal{A}) \le 2 \cdot \Bigg(& \frac{n_s^2 + n_r^2}{q} + \frac{n_r^2 + 3q_{\mathrm{RO}} \cdot n_r}{2^{160}} + \frac{q_{\mathrm{S}}}{2^{256}} \\ & + n_s n_r \cdot 1/\epsilon^{\mathsf{1succ}}_{\mathsf{X2Ell2}} \cdot \left(\mathsf{Adv}^{\mathsf{GapDH}}_{\mathsf{X25519}}(\mathcal{B}_1) + \frac{1}{2^{256}} \right) \\ & + (n_s + n_r) \cdot \Bigg(n_s \cdot (1/\epsilon^{\mathsf{1succ}}_{\mathsf{X2Ell2}})^2 \cdot \mathsf{Adv}^{\mathsf{GapDH}}_{\mathsf{X25519}}(\mathcal{B}_2) \\ & \qquad + \frac{3q_{\mathrm{RO}}}{2^{160}} + 2q_{\mathrm{C}} \cdot \mathsf{Adv}^{\mathsf{pk\text{-}unif}}_{\mathsf{X2Ell2}}(\mathcal{B}_3) \Bigg) \Bigg),\end{aligned}$$

where $\mathcal{A}$ *makes at most* q_{S}*,* q_{C}*, and* q_{RO} *queries to its* Send *oracle,* ChallExec *oracle, and the random oracle;* n_s*,* n_r *are the number sessions and servers (parties in* responder *role) that* $\mathcal{A}$ *interacts with; and* $q \approx 2^{252}$ *is the* X25519 *group order.* X2Ell2 *denotes the sample-and-encode (Definition 2.2) obfuscated key generation constructed from* X25519 *key generation and the* Elligator2 *encoding.*

Proof sketch. The proof proceeds via a series of game hops and branches. Our goal is to ensure that Finalize in Figure 5 returns a random bit, so we consider each clause in Finalize. We first rule out collisions in the DH shares (term $(n_s^2+n_r^2)/q$), which ensures soundness (Sound).

A first branch then rules out that $\mathcal{A}$ successfully probes a server (Probed): after excluding *NodeID* collisions ($n_r^2/2^{160}$) and ruling out $\mathcal{A}$ querying the HMAC random oracle on an uncompromised *NodeID* value ($3q_{\mathrm{RO}} \cdot n_r/2^{160}$), the client MAC MAC_C in a probing attempt is a random 256-bit value leaving $\mathcal{A}$ a $q_{\mathrm{S}}/2^{256}$ guessing chance.

A second branch prevents explicit authentication (ExplicitAuth) being violated: guessing the first session violating ExplicitAuth and its peer (losing a factor $n_s \cdot n_r$), and ruling out the case that its DH share is not encodable (losing a factor $1/\epsilon^{\mathsf{1succ}}_{\mathsf{X2Ell2}}$) , we can embed a GapDH challenge in that session's X DH public and the partner's B DH public key (term $\mathsf{Adv}^{\mathsf{GapDH}}_{\mathsf{X25519}}(\mathcal{B}_1)$) resulting in a truly random DH shared secret; after that, $\mathcal{A}$ only has a $1/2^{256}$ chance to guess the correct *auth* value violating explicit authentication for that session.

We then restrict the adversary to a single-challenge (rObfKE-1*) version of the game, applying the hybrid argument from Section 4.2.4 with a $(n_s + n_r)$ loss. We separately treat the following two cases.

For Case I ($\mathcal{A}$ makes a single Test query), we guess the (*sid*- or *cid*-)partner session of the tested session (n_s loss) and abort if the DH share in either session is not encodable ($(1/\epsilon^{\mathsf{1succ}}_{\mathsf{X2Ell2}})^2$ loss). We then embed a GapDH challenge into the X and Y DH public keys in the test session and its partner session (term $\mathsf{Adv}^{\mathsf{GapDH}}_{\mathsf{X25519}}(\mathcal{B}_2)$), resulting in the test session's key being random, which concludes this case.

For Case II ($\mathcal{A}$ wins via ChallExec queries to a single server), we first rule out that $\mathcal{A}$ queries the HMAC random oracle on the uncompromised *NodeID* belong to the single challenged server (term $3q_{\mathrm{RO}}/2^{160}$). Rewriting the DH inputs to HMAC, we then sample X' and Y' directly as random strings and decode them to DH public keys X and Y, instead of normally generating them; this can be bounded by the uniformity of X25519 public keys encoded with Elligator2 via a hybrid argument of all q_{C} many ChallExec queries, i.e., $2q_{\mathrm{C}} \cdot \mathsf{Adv}^{\mathsf{pk\text{-}unif}}_{\mathsf{X2Ell2}}(\mathcal{B}_3)$. At this point, the protocol messages in ChallExec sessions are uniformly distributed like outputs of the simulator $\mathcal{S}_{\mathtt{obfs4}}$, concluding this case and the proof. □

6 The `st-obfs` Protocol with Strong Obfusation

In `obfs4`, MAC tags are computed using HMAC keyed with the long-term public key of the server, B and *NodeID*. As a result, should the server's long-term public key leak to an adversary, the adversary can immediately identify prior and future traffic to this particular server.[4] We present a simple variant of the `obfs4` protocol, which we call `st-obfs`, which achieves our notion of *strong obfuscation*: providing obfuscation as long as the server's *secret* key is unrevealed; the public key however may be revealed to the adversary. In practice this means that an adversary cannot leverage obtained Tor bridge information (B, *NodeID*) to identify traffic to/from the bridge server.

Our `st-obfs` protocol is presented in Figure 6. In comparison to `obfs4`, it replaces the long-term public information used as the key to HMAC with the ephemeral/long-term DH key $\mathsf{DH}(X, B)$—a key which is also computed in `obfs4`, just that here we derive and use it earlier. We remark that the changes require that the server decodes the encoded public key X' and compute the ephemeral handshake key X^b before being able to verify the MAC tags. This increases the workload on the server by one decoding and one modular exponentiation operation before it can rule out any probing attempt.

Security. We establish sObfKE security for `st-obfs` under the same assumptions used in our result for `obfs4` (cf. Theorem 5.1). The simulator $\mathcal{S}_{\mathtt{obfs4}}$ as well as the session and contributive identifiers are defined as for `obfs4` (cf. Section 5).

[4] This property has previously been noted by David Fifield: https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/lyrebird/-/issues/30716#note_2832771.

```
Server key generation/setup
$NodeID \xleftarrow{\$} \{0,1\}^{160}$
$(b, B) \xleftarrow{\$} \mathsf{KGen}_{\mathsf{X25519}}()$
$st.\mathcal{S}_{MAC} \leftarrow \emptyset$
return $((b, NodeID), (B, NodeID), st)$

Client  knows $(B, NodeID)$                                   knows $(b, NodeID)$  Server
$(x, X, X') \xleftarrow{\$} \mathsf{X2Ell2.KGen}()$
$P_C \xleftarrow{\$} \{0,1\}^{\mathsf{pmin_C..pmax_C}}$
$M_C \leftarrow \mathsf{HMAC}(B^x, B \| NodeID \| X')$
$MAC_C \leftarrow \mathsf{HMAC}(B^x, B \| NodeID \| X' \| P_C \| M_C)$
            $\xrightarrow{msg_C = X' \| P_C \| M_C \| MAC_C}$
            $X' \leftarrow msg_C[1..\mathrm{ol}]$
            $X \leftarrow \mathsf{X2Ell2.Decode}(X')$
            $M_C \leftarrow \mathsf{HMAC}(X^b, B \| NodeID \| X')$
            parse $(X' \| P_C \| M_C \| MAC_C) \leftarrow msg_C$ using $M_C$; else break
            if $\mathsf{HMAC}(X^b, B \| NodeID \| X' \| P_C \| M_C) \neq MAC_C$: break
            if $MAC_C \in st.\mathcal{S}_{MAC}$: break
            $st.\mathcal{S}_{MAC} \leftarrow st.\mathcal{S}_{MAC} \cup \{MAC_C\}$
            $(y, Y, Y') \xleftarrow{\$} \mathsf{X2Ell2.KGen}()$
            // ntor handshake component
            $protoID \leftarrow$ “strong-obfs”
            $secret\_input \leftarrow X^y \| X^b \| B \| B \| X \| Y \| protoID \| NodeID$
            $skey \leftarrow \mathsf{HMAC}(secret\_input,$ “:key_extract”$)$
            $verify \leftarrow \mathsf{HMAC}(secret\_input,$ “:key_verify”$)$
            $auth\_input \leftarrow verify \| B \| B \| X \| Y \| protoID \| NodeID \|$“Server”
            $auth \leftarrow \mathsf{HMAC}(auth\_input,$ “:mac”$)$
            // end ntor handshake component
            $P_S \xleftarrow{\$} \{0,1\}^{\mathsf{pmin_S..pmax_S}}$
            $M_S \leftarrow \mathsf{HMAC}(X^b, B \| NodeID \| Y')$
            $MAC_S \leftarrow \mathsf{HMAC}(X^b, B \| NodeID \| Y' \| auth \| P_S \| M_S)$
            $\xleftarrow{msg_S = Y' \| auth \| P_S \| M_S \| MAC_S}$
$Y' \leftarrow msg_S[1..\mathrm{ol}]$
$M_S \leftarrow \mathsf{HMAC}(B^x, B \| NodeID, Y')$
parse $(Y' \| auth \| P_S \| M_S \| MAC_S) \leftarrow msg_S$ using $M_S$; else break
if $\mathsf{HMAC}(B^x, B \| NodeID \| Y' \| auth \| P_S \| M_S) \neq MAC_S$: break
$Y \leftarrow \mathsf{X2Ell2.Decode}(Y')$
// ntor handshake component
$protoID \leftarrow$ “strong-obfs”
$secret\_input \leftarrow Y^x \| B^x \| B \| B \| X \| Y \| protoID \| NodeID$
$skey \leftarrow \mathsf{HMAC}(secret\_input,$ “:key_extract”$)$
$verify \leftarrow \mathsf{HMAC}(secret\_input,$ “:key_verify”$)$
$auth\_input \leftarrow verify \| B \| B \| X \| Y \| protoID \| NodeID \|$“Server”
if $\mathsf{HMAC}(auth\_input,$ “:mac”$) \neq auth$: break
```

Figure 6: Our obfuscated key exchange protocol `st-obfs` with strong obfuscation. Changes compared to the original `obfs4` protocol (cf. Figure 4) are highlighted in blue.

Theorem 6.1. *Let* `st-obfs` *be defined as in Figure 6. Assume the* GapDH *problem is hard for* X25519 *and that* HMAC *behaves like a random oracle. For any* sObfKE *adversary* $\mathcal{A}$ *against* `st-obfs`*, we give algorithms* $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4$ *in the proof such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{sObfKE}}_{\mathtt{st\text{-}obfs},\mathcal{S}_{\mathtt{obfs4}}}(\mathcal{A}) \leq 2 \cdot \Bigg(& \frac{n_s^2 + n_r^2}{q} + \frac{n_r^2 + 3q_{\mathrm{RO}} \cdot n_r}{2^{160}} + \frac{q_{\mathrm{S}}}{2^{256}} \\ & + n_s n_r \cdot 1/\epsilon^{\mathsf{1succ}}_{\mathsf{X2Ell2}} \cdot \left(\mathsf{Adv}^{\mathsf{GapDH}}_{\mathsf{X25519}}(\mathcal{B}_1) + \frac{1}{2^{256}}\right) \\ & + (n_s + n_r \cdot q_{\mathrm{C}}) \cdot \bigg(n_s \cdot (1/\epsilon^{\mathsf{1succ}}_{\mathsf{X2Ell2}})^2 \cdot \mathsf{Adv}^{\mathsf{GapDH}}_{\mathsf{X25519}}(\mathcal{B}_2) \\ & \quad + (1/\epsilon^{\mathsf{1succ}}_{\mathsf{X2Ell2}})^2 \cdot \left(\mathsf{Adv}^{\mathsf{GapDH}}_{\mathsf{X25519}}(\mathcal{B}_3) + 2 \cdot \mathsf{Adv}^{\mathsf{pk\text{-}unif}}_{\mathsf{X2Ell2}}(\mathcal{B}_4)\right)\bigg)\Bigg),\end{aligned}$$

where $\mathcal{A}$ *makes at most* q_{S}, q_{C}, *and* q_{RO} *queries to its* SEND *oracle,* CHALLEXEC *oracle, resp. the random oracle,* n_s, n_r *are the number sessions, resp. servers (parties in* responder *role) that* $\mathcal{A}$ *interacts with, and* $q \approx 2^{252}$ *is the* X25519 *group order.*

We defer the proof to Appendix E.

6.1 Forward Obfuscation

Our notion of *strong obfuscation* differs from that of *forward secrecy*. Forward secrecy requires that an adversary cannot learn past session keys even if they have compromised a secret key. An analogous *forward obfuscation* property can be defined: an adversary cannot distinguish past traffic from simulated traffic even if a secret key is revealed. Our construction does not, however, achieve such a property. Strong obfuscation ensures that traffic is indistinguishable from simulated traffic when *public keys* are revealed, yet it does not protect in the setting where a *secret key* is revealed. In particular, if the secret key of a server is revealed (in `obfs4`, this is b and *NodeID*), then an adversary is able to distinguish prior traffic. This is true both for the existing `obfs4` protocol and our `st-obfs` variant. In theory, one could introduce the property of forward obfuscation into the protocol by employing puncturing pseudorandom functions, as in [8], or by using a time-based evolution of keys. However, the use of PPRFs is expensive and a time-based evolution of keys would only be of value in the case where bridge servers are relatively long-lived.

We collected data from the Collector's bridgepool assignment archive [48] to determine the approximate lifetime of a Tor bridge server and estimate the value of introducing forward obfuscation. Data was collected from between June to December 2023, from which we calculated statistics about the distribution of bridge lifetimes (capped at 6 months for each bridge server). Of the 4516 bridge servers that were active between June and December 2023, we observed the average lifetime of a bridge server to be 10.42 days and the 90th percentile to be 19.93 days. A large proportion of bridge servers are observed to be short-lived; nonetheless, bridge operators may consider stronger obfuscation properties desirable in some cases. We leave such constructions to future work.

7 Quantum-safe Obfuscated Key Exchange

In this section we present our quantum-safe obfuscated key exchange protocol `pq-obfs` with strong obfuscation.

7.1 The `pq-obfs` Protocol

Our `pq-obfs` protocol achieves each of the security goals outlined in Section 4.1: key indistinguishability, strong obfuscation, probing resistance, and explicit authentication. Additionally, the use of KEMs in place of the elliptic curve key exchange operations enables post-quantum security of the protocol. pq-obfs is largely modelled after obfs4 and ntor. At a high-level, we have replaced the ephemeral-static DH exchange in obfs4 with a static KEM encapsulation. Strong obfuscation is achieved via an ephemeral key input to the computations of MAC tags, similar to the approach in st-obfs (cf. Section 6). Finally, KEM public keys and ciphertexts sent between client and server must be indistinguishable from random. For this, we require that the protocol uses an obfuscated KEM (Definition 2.5). The complete protocol flow is given in Figure 7; Figure 9 in the appendix details the key schedule. We assume that, for replay protection, the protocol works in epochs, but we omit epochs from the description for simplicity (as we did for obfs4).

In our protocol, F_1 is a secure pseudorandom function with output length fl_1. A *PRF* $\mathsf{F}\colon \mathcal{X} \times \mathcal{Y} \to \mathcal{Z}$ treats the first input as a key and the second as a message, and for an unknown key should be indistinguishable from a random function on the message input. The *swap* of F is $\mathsf{F}'(y, x) = \mathsf{F}(x, y)$. We say that F is a *swap-PRF* if its swap F' is a PRF. We will treat F_2 as a *dual-PRF*, i.e., both a PRF and a swap-PRF, with output length fl_2. We may also refer to F_2 as a *key combiner*. See Appendix A for the details of PRF definitions.

Instantiating the protocol. Our construction is generic in the sense that any combination of an obfuscated KEM and functions F_1, F_2 can be used as long as they satisfy certain properties. In particular, we require an obfuscated KEM, as defined in Definition 2.5, that satisfies IND-CCA and SPR-CCA security. This is achieved by ML-Kemeleon (ML-KEM [46] with our Kemeleon encoding of public keys and ciphertexts as in Section 2.4). The IND-CCA and SPR-CCA security of ML-KEM, established in [15, 36, 43], translate to ML-Kemeleon, with a small loss from its first-sample/encapsulate success probability and ciphertext uniformity; see Theorems 2.12 and 2.13. We can instantiate F_1 and F_2 with HMAC, whose dual-PRF security for fixed-length keys is proven in [9].

7.2 Security Analysis of pq-obfs

We now analyze the security of pq-obfs in our model, showing that it achieves strong obfuscation (sObfKE) security. Our results rely on the IND-CCA and SPR-CCA security, public key and ciphertext uniformity, public key collision probability, and correctness of the obfuscated KEM OKEM, in addition to PRF security of F_1 and F_2, and swap-PRF security of F_2.

Simulator definition. We establish sObfKE security with respect to the following simulator $\mathcal{S}_{\text{pq-obfs}}$ which outputs two uniformly random messages of length corresponding to the pq-obfs message elements including random-length padding:

```
$\mathcal{S}_{\text{pq-obfs}}$:
$r_C \xleftarrow{\$} [\mathsf{pmin}_C, \mathsf{pmax}_C]$ ; $r_S \xleftarrow{\$} [\mathsf{pmin}_S, \mathsf{pmax}_S]$   // random-length client/server padding
$m_1 \xleftarrow{\$} \{0,1\}^{\mathsf{ol}+\mathsf{cl}+2\cdot\mathsf{fl}_1+r_C}$   // 1x encoded pk + 1x encoded ctxt + 2x $\mathsf{F}_1$ outputs + client padding
$m_2 \xleftarrow{\$} \{0,1\}^{\mathsf{cl}+3\cdot\mathsf{fl}_1+r_S}$   // 1x encoded ctxt + 3x $\mathsf{F}_1$ outputs + server padding
return $(m_1, m_2)$
```

Session and contributive identifiers. We set the session identifeir as

$$sid := (pk_e, c_e, pk_S, c_S),$$

where pk_e is the initiator's KEM public key, pk_S is the responder's static KEM public key, and *NodeID* is the responder's long-term identifier. We set the contributive identifier to

$$cid := (pk_e, c_S)$$

upon the client sending or server receiving the first message.

```
Server key generation/setup
$NodeID \xleftarrow{\$} \{0,1\}^{nl}$
$(pk_S, sk_S, \_) \xleftarrow{\$} \mathsf{OKEM.KGen}()$
$st.\mathcal{S}_{MAC} \leftarrow \emptyset$
return $((sk_S, NodeID), (pk_S, NodeID), st)$

Client  knows $(pk_S, NodeID)$                          knows $(sk_S, NodeID)$  Server
$(sk_e, pk_e, \widehat{pk}_e) \xleftarrow{\$} \mathsf{OKEM.KGen}()$
$P_C \xleftarrow{\$} \{0,1\}^{\mathsf{pmin}_C..\mathsf{pmax}_C}$
$(c_S, K_S, \widehat{c}_S) \xleftarrow{\$} \mathsf{OKEM.Encap}(pk_S)$
$ES \leftarrow \mathsf{F}_2(NodeID, K_S)$
$M_C \leftarrow \mathsf{F}_1(ES, \widehat{pk}_e \| \widehat{c}_S \| \text{“:mc”})$
$MAC_C \leftarrow \mathsf{F}_1(ES, \widehat{pk}_e \| \widehat{c}_S \| P_C \| M_C \| \text{“:mac\_c”})$
                 $\xrightarrow{msg_C = \widehat{pk}_e \| \widehat{c}_S \| P_C \| M_C \| MAC_C}$
        $\widehat{pk}_e \leftarrow msg_C[1..\mathsf{ol}]$ ; $\widehat{c}_S \leftarrow msg_C[\mathsf{ol}+1..\mathsf{ol}+\mathsf{cl}]$
        $K_S \leftarrow \mathsf{OKEM.Decap}(sk_S, \widehat{c}_S)$
        $ES \leftarrow \mathsf{F}_2(NodeID, K_S)$
        $M_C \leftarrow \mathsf{F}_1(ES, \widehat{pk}_e \| \widehat{c}_S \| \text{“:mc”})$
        parse $(\widehat{pk}_e \| \widehat{c}_S \| P_C \| M_C \| MAC_C) \leftarrow msg_C$ using $M_C$; else break
        if $\mathsf{F}_1(ES, \widehat{pk}_e \| \widehat{c}_S \| P_C \| M_C \| \text{“:mac\_c”}) \neq MAC_C$: break
        if $MAC_C \in st.\mathcal{S}_{MAC}$: break
        $pk_e \leftarrow \mathsf{OKEM.Decode}(\widehat{pk}_e)$
        $(c_e, K_e, \widehat{c}_e) \xleftarrow{\$} \mathsf{OKEM.Encap}(pk_e)$
        $protoID \leftarrow$ “pq-obfs”
        $ES' \leftarrow \mathsf{F}_1(ES, \text{“:derive\_key”})$ ; $FS \leftarrow \mathsf{F}_2(ES', K_e)$
        $context \leftarrow pk_S \| c_S \| pk_e \| c_e \| protoID$
        $skey \leftarrow \mathsf{F}_1(FS, context \| \text{“:key\_extract”})$
        $auth \leftarrow \mathsf{F}_1(FS, context \| \text{“:server\_mac”})$
        $P_S \xleftarrow{\$} \{0,1\}^{\mathsf{pmin}_S..\mathsf{pmax}_S}$
        $M_S \leftarrow \mathsf{F}_1(ES, \widehat{c}_e \| \text{“:ms”})$
        $MAC_S \leftarrow \mathsf{F}_1(ES, \widehat{c}_e \| auth \| P_S \| M_S \| \text{“:mac\_s”})$
                 $\xleftarrow{msg_S = \widehat{c}_e \| auth \| P_S \| M_S \| MAC_S}$
$\widehat{c}_e \leftarrow msg_S[1..\mathsf{cl}]$
$M_S \leftarrow \mathsf{F}_1(ES, \widehat{c}_e \| \text{“:ms”})$
parse $(\widehat{c}_e \| auth \| P_S \| M_S \| MAC_S) \leftarrow msg_S$ using $M_S$; else break
if $\mathsf{F}_1(ES, \widehat{c}_e \| auth \| P_S \| M_S \| \text{“:mac\_s”}) \neq MAC_S$: break
$K_e \leftarrow \mathsf{OKEM.Decap}(sk_e, \widehat{c}_e)$
$protoID \leftarrow$ “pq-obfs”
$ES' \leftarrow \mathsf{F}_1(ES, \text{“:derive\_key”})$ ; $FS \leftarrow \mathsf{F}_2(ES', K_e)$
$context \leftarrow pk_S \| c_S \| pk_e \| c_e \| protoID$
$skey \leftarrow \mathsf{F}_1(FS, context \| \text{“:key\_extract”})$
if $\mathsf{F}_1(FS, context \| \text{“:server\_mac”}) \neq auth$: break
```

Figure 7: Our post-quantum obfuscated key exchange protocol pq-obfs with strong obfuscation.

We now give the theorem statement and a proof sketch; the complete proof is deferred to Appendix F.

Theorem 7.1. *Let* pq-obfs *be defined as in Figure 7. For any* sObfKE *adversary* $\mathcal{A}$ *against* pq-obfs*, we give algorithms* $\mathcal{B}_1$–$\mathcal{B}_{17}$ *in the* *proof such that*

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{sObfKE}}_{\mathtt{pq\text{-}obfs},\mathcal{S}_{\mathtt{pq\text{-}obfs}}}(\mathcal{A}) \le 2 \cdot \Big(& \mathsf{pkcoll}_{\mathsf{OKEM}}(n_s + n_r) + 2n_s \cdot \delta_{\mathsf{OKEM}} \\
& + n_s n_r \cdot \Big(\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_2}(\mathcal{B}_1) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_2) + \frac{1}{2^{\mathsf{fl}_1}} \Big) \\
& + n_s n_r \cdot \Big(\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{OKEM}}(\mathcal{B}_3) + \mathsf{Adv}^{\mathsf{swap\text{-}PRF}}_{\mathsf{F}_2}(\mathcal{B}_4) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_2}(\mathcal{B}_5) \\
& \qquad + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_6) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_7) + \frac{1}{2^{\mathsf{fl}_1}} \Big) \\
& + (n_s + n_r q_{\mathrm{C}}) \cdot \Big(n_s \cdot \Big(\mathsf{Adv}^{\mathsf{IND\text{-}1CCA}}_{\mathsf{OKEM}}(\mathcal{B}_8) + \mathsf{Adv}^{\mathsf{swap\text{-}PRF}}_{\mathsf{F}_2}(\mathcal{B}_9) \\
& \qquad\qquad + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_{10}) \Big) \\
& \qquad + \mathsf{Adv}^{\mathsf{SPR\text{-}CCA}}_{\mathsf{OKEM}}(\mathcal{B}_{11}) + \mathsf{Adv}^{\mathsf{swap\text{-}PRF}}_{\mathsf{F}_2}(\mathcal{B}_{12}) \\
& \qquad + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_{13}) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_2}(\mathcal{B}_{14}) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_{15}) \\
& \qquad + \mathsf{Adv}^{\mathsf{ctxt\text{-}unif}}_{\mathsf{OKEM}}(\mathcal{B}_{16}) + \mathsf{Adv}^{\mathsf{pk\text{-}unif}}_{\mathsf{OKEM}}(\mathcal{B}_{17}) \Big) \Big),
\end{aligned}$$

where $\mathcal{A}$ makes at most q_{C} ChallExec *queries, n_s, n_r are the number sessions and servers (parties in* responder *role) that $\mathcal{A}$ interacts with, nl is the NodeID bit-length, and* fl_1 *is the output bit-length of* F_1.

Proof sketch. Similar to the `obfs4` proof (Theorem 5.1), we proceed via a series of game hops and branches, considering each clause in Finalize. We first rule out KEM public key collisions (term $\mathsf{pkcoll}_{\mathsf{OKEM}}(n_s + n_r)$) and assume correct decapsulation in all sessions ($2n_s \cdot \delta_{\mathsf{OKEM}}$), which ensures soundness (Sound).

A first branch then rules out that $\mathcal{A}$ successfully probes a server (Probed): we guess the first server and session that sets Probed (with a $n_s \cdot n_r$ loss). From that server's uncompromised *NodeID*, we apply PRF security to turn *ES* and MAC_C into random values, after which $\mathcal{A}$ must have guessed a random fl_1-bit MAC_C; the corresponding bound is $\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_2}(\mathcal{B}_1) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_2) + 1/2^{\mathsf{fl}_1}$.

A second branch prevents explicit authentication (ExplicitAuth) being violated: guessing the first session violating ExplicitAuth and its peer (losing a factor $n_s \cdot n_r$), we can embed an IND-CCA challenge in that the peer's public key, the session's c_S, and the derived K_S key (term $\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{OKEM}}(\mathcal{B}_3)$). Through four (swap-)PRF hops ($\mathcal{B}_4$–$\mathcal{B}_7$), we show that this turns *auth* into a random value leaving $\mathcal{A}$ with a $1/2^{\mathsf{fl}_1}$ chance to violate ExplicitAuth.

We then restrict the adversary to a single-challenge (sObfKE-1) version of the game, applying the hybrid argument from Section 4.2.4 with a $(n_s + n_r q_{\mathrm{C}})$ loss. We separately treat the following two cases.

For Case I ($\mathcal{A}$ makes a single Test query), we first guess the (*sid*- or *cid*-)partner session of the tested session (losing a factor n_s). We then embed an IND-1CCA challenge in the ephemeral KEM encapsulation of the test session and its partner session (term $\mathsf{Adv}^{\mathsf{IND\text{-}1CCA}}_{\mathsf{OKEM}}(\mathcal{B}_8)$), followed by two (swap-)PRF hops ($\mathcal{B}_9$, $\mathcal{B}_{10}$) to randomize the test session's key, concluding this case.

For Case II ($\mathcal{A}$ makes a single ChallExec query to a predetermined server), we embed a SPR-CCA challenge in the long-term KEM encapsulation of the ChallExec sessions (term $\mathsf{Adv}^{\mathsf{SPR\text{-}CCA}}_{\mathsf{OKEM}}(\mathcal{B}_{11})$), randomizing both $\widehat{c}_S$ and K_S. We proceed via four (swap-)PRF hops ($\mathcal{B}_{12}$–$\mathcal{B}_{15}$) to randomize the transcript's MAC values as well as the session key. Finally, we apply the ciphertext and public key uniformity of the obfuscated KEM OKEM (terms $\mathsf{Adv}^{\mathsf{ctxt\text{-}unif}}_{\mathsf{OKEM}}(\mathcal{B}_{16})$ and $\mathsf{Adv}^{\mathsf{pk\text{-}unif}}_{\mathsf{OKEM}}(\mathcal{B}_{17})$) to randomize $\widehat{c}_e$ and $\widehat{pk}_e$. At this point, the protocol messages in ChallExec sessions are uniformly distributed like outputs of the simulator $\mathcal{S}_{\mathtt{pq\text{-}obfs}}$, concluding this case and the proof. □

8 Discussion and Further Challenges

8.1 Simulating TLS1.3

Due to the pervasiveness of TLS [50], circumvention tools often attempt to mimic TLS patterns [29] to hide within a large anonymity set. It is hence natural to consider if one could prove that a particular obfuscated key exchange protocol is indistinguishable from a TLS handshake in our model.

TLS 1.3 [50] is not inherently resistant to active probing: there is no mechanism for a server to distinguish between honest clients and probing attempts. This is because public keys are assumed to be public, unlike the semi-private bridge server information in Tor. Nonetheless, we can consider how to simulate a TLS 1.3 handshake. The simulator is given a coroutine that defines TLS 1.3 behaviour and distribution of handshake fields. To achieve probing resistance, an obfuscated protocol mimicing TLS 1.3 should copy the same failure modes. Our model currently does not capture responses to probes other than empty messages, but this is easily generalized to reflect an alternative error response to an unverified probe.

TLS 1.3 handshakes have the following fields in which one might embed additional information:

- **Nonce**: The 256-bit random nonce field can be used to embed additional information, e.g., an X25519 key as discussed in [26].
- **Ephemeral Public Key**: One can of course send a public key in the field intended to contain a public key, but any value following the same distribution as the appropriate public key (e.g. X25519, X448, depending on the version) would suffice.
- **PSK Binder Value**: Binder values typically contain the output of a MAC applied to a pre-shared key identifier and nonce. One could embed in this field anything following the same distribution as the output of the corresponding MAC function, e.g., a MAC of semi-secret bridge information for probing resistance.
- **PSKIdentity**: The identity values of pre-shared keys are chosen by servers an may be arbitrarily distributed; however, uniformly random strings would be a natural choice, such as in the case of encrypted TLS session tickets.

These fields need not necessarily be uniformly random. For example, in Cloak [62], which composes a TLS connection and hides information within the handshake, ephemeral key information can be embedded in a field intended for public keys. This avoids any issues with distinguishability due to keys not being uniformly random.[5]

We provide a theoretical foundation for correctly embedding information in various TLS handshake fields. However, as pointed out by [29, 37], staying up-to-date with current TLS implementations to avoid fingerprinting of protocols can be difficult in practice.

8.2 Defining a Simulator

Obfuscation security is proven in our model with respect to a specific simulator, capturing the class of protocols that a protocol hides within. One might imagine an ideal setting in which all protocols

[5] Such as https://github.com/net4people/bbs/issues/287#issuecomment-1718920382.

fall into the same class and thus, all handshakes could be proven indistinguishable from another. In reality, protocols have a surfeit of properties which distinguish them from one another. Capturing all of these properties in a singular definition or model is challenging, and defining a concrete collection of features that, when combined, is hard to fingerprint remains an open problem. Defining security with respect to a given simulator makes our model flexible and avoids prescribing a (potentially flawed) selection of features.

Towards determining what collection of features are relevant for obfuscated protocols (and defining simulators), we perform a brief literature review of work that takes a feature-based detection approach to classifying obfuscated traffic. Many prior works develop machine-learning-based techniques for classifying obfuscated traffic. These models are typically trained on a set of features that are useful in fingerprinting such protocols. A systematic collection of these features provides a lower bound on what properties must be protected/obfuscated to avoid this class of blocking, even if not all of these features are used in practice for classifying traffic today.

Our review includes observations from both censorship circumvention research aiming to improve the state-of-the-art protocols and research developing censorship tools to block these protocols. We collect features from work which aims to classify obfuscated traffic (e.g., `obfs4`, Shadowsocks), underlying applications of encrypted traffic, and other circumvention tools (e.g., Snowflake [14]). Not all of the features we found in the literature can be captured by our model; we discuss challenges that remain in modeling them. We summarize the results of our review below; more details are given in Table 3.

The most commonly referenced features that are used for protocol classification are entropy, directionality, and length of packets. Each of these can be captured in our model through an according simulator. Notably, there is evidence that the GFW has blocked fully encrypted traffic with uncommonly high entropy [66]. This can generally be remedied through post-processing of the encrypted data to reduce randomness. Existing FEPs aim to obfuscate packet lengths, but do not prescribe any specific traffic pattern that would obfuscate the directionality of packets. Defining such effectively obfuscated directionality patterns remains an open problem.

Several works point out that analyzing protocols on a per-session or per-host basis can be more effective than analyzing individual packets [35, 42, 61]. Our model captures an adversary's ability to evaluate the key exchange protocol as a whole, and not necessarily individual packets; however, it does not consider how this analysis may be combined with the following data transfer phase. One might consider how our model can be composed with that of [24] to view the protocol as a whole. We note, however, that evidence of real-world blocking [66] suggested that only rudimentary classification tests were placed on the first TCP payload.

The features found in the literature that cannot be captured by our model are all properties related to *timing*: inter-packet delay, packet direction/timing sequence, total execution time, volume of traffic, and timeouts. The remaining features we found can all be captured in the definition of a simulator (or via probing resistance). Fenske and Johnson [24] include termination of TCP connections into their model of the data transfer phase and suggest that ideal behaviour for a FEP is to never close a connection. This addresses the timeout feature, but the remaining timing-based properties of packets remain unresolved. For example, the `obfs4` handshake requires that the client wait for the server's response before sending more data. This stems from the fact that any data following the handshake must be encrypted with the established shared key. This packet sequence is identifiable because, for the period of time that the client is waiting for the server's response, it sends no data.[6] Our current model is incapable of capturing this property because we model the handshake as a transcript of messages. Modeling timing behaviour is challenging because it requires a streaming-like notion [27] for key exchange protocols. Expanding key exchange models in general and our model in particular to capture timing properties of protocols is an interesting avenue for future work.

Acknowledgements

We would like to thank those from the anti-censorship research community who compiled many of the resources referenced in Section 8.2. We also thank David Fifield and Lindsey Tulloch for useful discussions on `obfs4`, Kien Tuong Truong for proposing the name Kemeleon, and Stanislaw Jarecki for insightful discussions about covert authenticated key exchange.

This work was supported by Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery grant RGPIN-2022-03187 and NSERC Alliance grant ALLRP 578463-22.

References

[1] eMule protocol obfuscation. https://wiki.emule-web.de/Protocol_obfuscation, 2010.

[2] Message stream encryption. https://web.archive.org/web/20150212025828/https://wiki.vuze.com/w/Message_Stream_Encryption, 2015.

[3] VMess. https://www.v2ray.com/en/configuration/protocols/vmess.html, 2019.

[4] Shadowsocks. https://shadowsocks.org/doc/what-is-shadowsocks.html, 2023.

[5] Alice, Bob, Carol, Jan Beznazwy, and Amir Houmansadr. How China detects and blocks Shadowsocks. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20, pages 111–124, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Anonymous, Kevin Bock, Jsckson Sippe, Shelikhoo, David Fifield, Eric Wustrow, Dave Levin, and Amir Houmansadr. Exposing the Great Firewall's dynamic blocking of fully encrypted traffic. Technical report, Open Technology Fund, 2022.

[7] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber algorithm specifications and supporting documentation (v3.02), 2021.

[8] Nimrod Aviram, Kai Gellert, and Tibor Jager. Session resumption protocols and efficient forward security for TLS 1.3 0-RTT. *Journal of Cryptology*, 34(3):20, July 2021.

[9] Matilda Backendal, Mihir Bellare, Felix Günther, and Matteo Scarlata. When messages are keys: Is HMAC a dual-PRF? In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part III*, volume 14083 of *LNCS*, pages 661–693. Springer, Heidelberg, August 2023.

[10] Mihir Bellare, Alexandra Boldyreva, and Silvio Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 259–274. Springer, Heidelberg, May 2000.

[11] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–155. Springer, Heidelberg, May 2000.

[12] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 232–249. Springer, Heidelberg, August 1994.

[13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 967–980. ACM Press, November 2013.

[6] This has previously been discussed on the tor-dev mailing list, see https://lists.torproject.org/pipermail/tor-dev/2017-June/012310.html.

Feature	Description	Source(s)
Entropy and Directionality		
— Entropy	Statistics (mean, max, min, etc.) on per-packet and overall randomness, e.g., Shannon entropy.	[33, 34, 41, 53, 61, 63, 66, 66]
— Printable characters	Statistics on number/proportion/existence of printable characters.	[6]
— Directionality	Statistics on number and ordering of packets in each direction (total, variance, entropy, ratio).	[19, 32, 34, 41, 65, 69, 70]
Packet length		
— Packet length	Statistics on per-packet length, distribution of packet lengths, and patterns in sequences of packet lengths.	[32–34, 41, 42, 63, 65, 69, 70]
Plaintext features		
— Message type sequence	For (D)TLS-based protocols, sequences of message types, e.g., Client Hello, Application Data, New Session Ticket, Finished, etc.	[42]
— Plaintext header/handshake features	For (D)TLS protocols, different implementations use different settings for properties such as record version, fragment offset, ciphersuite, extension length, extensions, etc.	[16, 42, 66]
Timing properties		
— Inter-packet delay	Statistics on time delay between packets.	[21, 32, 33, 63, 69]
— Packet timing sequence	For example, down-time after a client-to-server message while client waits for server response, as in `obfs4`.	[34, 41]
— Total time	Total duration of packet stream before connection is closed or becomes idle.	[19, 32]
— Volume of traffic	Bytes and number of packets per second.	[19, 32]
— Timeout	Time to close a connection after failed probing attempt.	[24, 28]
Probing resistance		
— Probing response	Properties of (and/or existence of) response to probing attempt.	[5, 28]
— Buffer threshold	Number of bytes (e.g., of probing attempt) read before closing a connection.	[24, 28]

Table 3: Summary of protocol features used by censors to detect of obfuscated protocols. References written only in Chinese were translated using DeepL. Only the features under the *Timing properties* category are *not* captured by our model.

[14] Cecylia Bocovich, Arlo Breault, David Fifield, Serene, and Ziaokang Wang. Snowflake, a censorship circumvention system using temporary WebRTC proxies. In *33nd USENIX Security Symposium, USENIX Security 2024*, 2024. To appear.

[15] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - Kyber: A CCA-secure module-lattice-based KEM. In *IEEE European Symposium on Security and Privacy, EuroS&P 2018*, pages 353–367. IEEE, 2018.

[16] Junqiang Chen, Guang Cheng, and Hantao Mei. F-ACCUMUL: A protocol fingerprint and accumulative payload length sample-based Tor-Snowflake traffic-identifying framework. *Applied Sciences*, 13(1), 2023.

[17] Katriel Cohn-Gordon, Cas Cremers, Kristian Gjøsteen, Håkon Jacobsen, and Tibor Jager. Highly efficient key exchange protocols with optimal tightness. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 767–797. Springer, Heidelberg, August 2019.

[18] Culture Club. Karma Chameleon. In *Colour by Numbers*, 1983.

[19] Ziye Deng, Zihan Liu, Zhouguo Chen, and Yubin Guo. The random forest based detection of Shadowsock's traffic. In *2017 9th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*, volume 2, pages 75–78, 2017.

[20] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 1197–1210. ACM Press, October 2015.

[21] Jie Du, Yongzhong He, and Ye Du. Improved method of Tor network flow watermarks based on IPD interval. *Chinese Journal of Network and Information Security*, 5(4):91–98, 2019.

[22] Karim Eldefrawy, Nicholas Genise, and Stanislaw Jarecki. Short concurrent covert authenticated key exchange (short cAKE). In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part VIII*, volume 14445 of *LNCS*, pages 75–109. Springer, Heidelberg, December 2023.

[23] Roya Ensafi, David Fifield, Philipp Winter, Nick Feamster, Nicholas Weaver, and Vern Paxson. Examining how the Great Firewall discovers hidden circumvention servers. In *Proceedings of the 2015 Internet Measurement Conference*, pages 445–458, 2015.

[24] Ellis Fenske and Aaron Johnson. Security notions for fully encrypted protocols. Free and Open Communications on the Internet 2023 Workshop, Issue 1, https://www.petsymposium.org/foci/2023/foci-2023-0004.php, 2023.

[25] Ellis Fenske and Aaron Johnson. Bytes to schlep? Use a FEP: Hiding protocol metadata with fully encrypted protocols. https://arxiv.org/pdf/2405.13310, 2024.

[26] Marc Fischlin. Stealth key exchange and confined access to the record protocol data in TLS 1.3. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023*, pages 2901–2914. ACM Press, November 2023.

[27] Marc Fischlin, Felix Günther, Giorgia Azzurra Marson, and Kenneth G. Paterson. Data is a stream: Security of stream-based channels. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 545–564. Springer, Heidelberg, August 2015.

[28] Sergey Frolov, Jack Wampler, and Eric Wustrow. Detecting probe-resistant proxies. In *NDSS 2020*. The Internet Society, February 2020.

[29] Sergey Frolov and Eric Wustrow. The use of TLS in censorship circumvention. In *NDSS 2019*. The Internet Society, February 2019.

[30] Ian Goldberg. [tor-dev] RFC on obfs3 pluggable transport. https://lists.torproject.org/pipermail/tor-dev/2012-December/004245.html, 2012.

[31] Ian Goldberg, Douglas Stebila, and Berkant Ustaoglu. Anonymity and one-way authentication in key exchange protocols. *DCC*, 67(2):245–269, 2013.

[32] Zheyuan Gu, Gaopeng Gou, Chengshang Hou, Gang Xiong, and Zhen Li. LFETT2021: A large-scale fine-grained encrypted tunnel traffic dataset. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 240–249, 2021.

[33] Zhong Guan, Gaopeng Gou, Yangyang Guan, and Bingxu Wang. An empirical analysis of plugin-based Tor traffic over SSH tunnel. In *MILCOM 2019 - 2019 IEEE Military Communications Conference (MILCOM)*, pages 616–621, 2019.

[34] Yongzhong He, Liping Hu, and Rui Gao. Detection of Tor traffic hiding under obfs4 protocol based on two-level filtering. In *2019 2nd International Conference on Data Intelligence and Security (ICDIS)*, pages 195–200, 2019.

[35] Erik Hjelmvik and Wolfgang John. Breaking and improving protocol obfuscation. Technical Report 2010-05, ISSN 1652-926X, Department of Computer Science and Engineering, Chalmers University of Technology, 2010.

[36] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 341–371. Springer, Heidelberg, November 2017.

[37] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The parrot is dead: Observing unobservable network communications. In *2013 IEEE Symposium on Security and Privacy*, pages 65–79. IEEE Computer Society Press, May 2013.

[38] Stanislaw Jarecki. Practical covert authentication. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 611–629. Springer, Heidelberg, March 2014.

[39] T. Kivinen and M. Kojo. More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). RFC 3526 (Proposed Standard), May 2003.

[40] A. Langley, M. Hamburg, and S. Turner. Elliptic Curves for Security. RFC 7748 (Informational), January 2016.

[41] Di Liang and Yongzhong He. Obfs4 traffic identification based on multiple-feature fusion. In *2020 IEEE International Conference on Power, Intelligent Computing and Systems (ICPICS)*, pages 323–327, 2020.

[42] Chang Liu, Gang Xiong, Gaopeng Gou, Siu-Ming Yiu, Zhen Li, and Zhihong Tian. Classifying encrypted traffic using adaptive fingerprints with multi-level attributes. *World Wide Web*, 24(6):2071–2097, 2021.

[43] Varun Maram and Keita Xagawa. Post-quantum anonymity of Kyber. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part I*, volume 13940 of *LNCS*, pages 3–35. Springer, Heidelberg, May 2023.

[44] Bodo Möller. A public-key encryption scheme with pseudo-random ciphertexts. In Pierangela Samarati, Peter Y. A. Ryan, Dieter Gollmann, and Refik Molva, editors, *ESORICS 2004*, volume 3193 of *LNCS*, pages 335–351. Springer, Heidelberg, September 2004.

[45] Y. Nir, T. Kivinen, P. Wouters, and D. Migault. Algorithm Implementation Requirements and Usage Guidance for the Internet Key Exchange Protocol Version 2 (IKEv2). RFC 8247 (Proposed Standard), September 2017. Updated by RFC 9395.

[46] NIST. Module-lattice-based key-encapsulation mechanism standard, August 2023. FIPS 203 (draft). https://doi.org/10.6028/NIST.FIPS.203.ipd.

[47] Tatsuaki Okamoto and David Pointcheval. The gap-problems: A new class of problems for the security of cryptographic schemes. In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 104–118. Springer, Heidelberg, February 2001.

[48] The Tor Project. Collector: Bridge pool assignments. https://collector.torproject.org/archive/bridge-pool-assignments/, 2023.

[49] The Tor Project. lyrebird, version 348eddc8. https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/lyrebird, 2023.

[50] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard), August 2018.

[51] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. TLS Encrypted Client Hello. https://www.ietf.org/archive/id/draft-ietf-tls-esni-14.html, 2022.

[52] Benjamin Schwartz and Christopher Patton. The pseudorandom extension for cTLS. https://www.ietf.org/archive/id/draft-cpbs-pseudorandom-ctls-01.html, 2022.

[53] Tang Shuye, Cheng Guang, Jiang Bomiao, Chen Zihan, and Guo Shuyi. Detection and recognition of VPN encrypted traffic based on segmented entropy distribution. *Cyberspace Security*, 11(8):23–33, 2020.

[54] The Tor Project. obfs3 (the threebfuscator), protocol specification, version 225e420c. https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/obfsproxy/-/blob/HEAD/doc/obfs3/obfs3-protocol-spec.txt, 2013.

[55] The Tor Project. obfs2 (the twobfuscator), protocol specification, version 2bf9d096. https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/obfsproxy/-/blob/master/doc/obfs2/obfs2-protocol-spec.txt, 2015.

[56] The Tor Project. obfs4 (the obfourscator), protocol specification, version c0898c2d. https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/lyrebird/-/blob/main/doc/obfs4-spec.txt, 2019.

[57] The Tor Project. Tor metrics: Bridge users by transport. https://metrics.torproject.org/userstats-bridge-transport.html?start=2024-01-01&end=2024-05-01&transport=!%3COR%3E&transport=obfs4, 2024.

[58] Mehdi Tibouchi. Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 139–156. Springer, Heidelberg, March 2014.

[59] Michael Carl Tschantz, Sadia Afroz, anonymous, and Vern Paxson. SoK: Towards grounding censorship circumvention in empiricism. In *2016 IEEE Symposium on Security and Privacy*, pages 914–933. IEEE Computer Society Press, May 2016.

[60] Luis von Ahn and Nicholas J. Hopper. Public-key steganography. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 323–341. Springer, Heidelberg, May 2004.

[61] Ryan Wails, George Arnold Sullivan, Micah Sherr, and Rob Jansen. On precisely detecting censorship circumvention in real-world networks. In *Network and Distributed System Security (NDSS) Symposium 2024*, 2024.

[62] Andy Wang. Cloak. https://github.com/cbeuw/Cloak/wiki/Steganography-and-encryption, 2022.

[63] Liang Wang, Kevin P. Dyer, Aditya Akella, Thomas Ristenpart, and Thomas Shrimpton. Seeing through network-protocol obfuscation. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 57–69. ACM Press, October 2015.

[64] Philipp Winter, Tobias Pulls, and Juergen Fuss. ScrambleSuit: A polymorphic network protocol to circumvent censorship. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pages 213–224, 2013.

[65] Hua Wu, Shuyi Guo, Guang Cheng, and Xiaoyan Hu. Detecting Tor bridge from sampled traffic in backbone networks. *Workshop on Measurements, Attacks, and Defenses for the web (MADWeb) 2021*, 2021.

[66] Mingshi Wu, Jackson Sippe, Danesh Sivakumar, Jack Burg, Peter Anderson, Xiaokang Wang, Kevin Bock, Amir Houmansadr, Dave Levin, and Eric Wustrow. How the Great Firewall of China detects and blocks fully encrypted traffic. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023*, pages 2653–2670. USENIX Association, 2023.

[67] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. Telex: Anticensorship in the network infrastructure. In *USENIX Security 2011*. USENIX Association, August 2011.

[68] Keita Xagawa. Anonymity of NIST PQC round 3 KEMs. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part III*, volume 13277 of *LNCS*, pages 551–581. Springer, Heidelberg, May / June 2022.

[69] Yibo Xie, Gaopeng Gou, Gang Xiong, Zhen Li, and Mingxin Cui. Covertness analysis of Snowflake proxy request. In *2023 26th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 1802–1807, 2023.

[70] Diwen Xue, Michalis Kallitsis, Amir Houmansadr, and Roya Ensafi. Fingerprinting obfuscated proxy traffic with encapsulated TLS handshakes. In *33nd USENIX Security Symposium, USENIX Security 2024*, 2024. To appear.

A Additional Definitions

Definition A.1 (Pseudorandom function). *A* pseudorandom function $\mathsf{F} : \{0,1\}^\kappa \times \{0,1\}^\iota \to \{0,1\}^\lambda$ *takes as input a key of length* κ *and an input of length* ι*, and produces an output of length* λ*.*

Definition A.2 (Secure pseudorandom function). *A pseudorandom function* F *is said to be* secure *if it is computationally infeasible for an adversary to distinguish the output of a pseudorandom function (under an unknown key) from the output of a random function. More precisely, let* $\mathsf{F} : \{0,1\}^\kappa \times \{0,1\}^\iota \to \{0,1\}^\lambda$ *and define*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}}(\mathcal{A}) &:= \Pr\left[\mathcal{A}^{\mathsf{F}(k,\cdot)}() \Rightarrow 1 \mid k \xleftarrow{\$} \{0,1\}^\kappa\right] \\ &\quad - \Pr\left[\mathcal{A}^{R(\cdot)}() \Rightarrow 1 \mid R \xleftarrow{\$} \{\textit{all functions} : \{0,1\}^\iota \to \{0,1\}^\lambda\}\right].\end{aligned}$$

B Public Key and Ciphertext Encodings

We provide the first systematic treatment of existing encodings between public keys and random strings. The `obfs4` construction makes use of the Elligator2 encoding. Before Elligator2, the first examples of mappings of public keys to random strings were the Uniform DH (UDH) encoding [30] and Telex's original-or-twist strategy [67]. We can also consider an aspect of the authenticated key exchange scheme from von Ahn and Hopper [60] as one such mapping. Here we review the details of each mapping and evaluate them according to the properties defined in Section 2. We summarize the properties of these mappings, in addition to our new encoding scheme Kemeleon, in Table 2.

von Ahn–Hopper [60]. In a construction for a steganogrpahic key exchange, von Ahn and Hopper [60, §6.3] propose a DH key generation algorithm that can be viewed as an encoding of DH public keys to random strings. Aligned with the sample-and-encode paradigm described in Section 2.1, public keys are sampled and then rejected if they do not satisfy an "encodable" form. In this case, working in the group $\mathbb{Z}_p$, the algorithm samples a private key $a \xleftarrow{\$} \mathbb{Z}_{p-1}$ and rejects the resulting public key, g^a for a generator g, if the most significant bit is 1. Else, it outputs all bits of g^a except the most significant bit (this bit is deterministically set to 0 upon decoding). Naturally, the resulting output is a perfectly uniform bitstring of length $\lceil \log_2(p) \rceil - 1$. The first-sample success probability is the probability that for a randomly sampled public key, its most significant bit is 0. This is $2^{\lceil \log_2(p) \rceil - 1}/p < \frac{1}{2}$. Pseudocode for the algorithm is given in Figure 8 and details are summarized in Table 2.

UDH. The Uniform DH map, originally proposed in [30], maps elements from a group $\mathbb{Z}_p$ to random values in the range $[0, 2^{|p|} - 1]$. In particular, the original proposal focuses on modular exponential groups designed for key exchange [39]. In the protocol description, the input is assumed to be an element of $\mathbb{Z}_p$ where $p \equiv 3 \bmod 4$ is prime with $q = (p-1)/2$ also prime. We also require that p is close to a multiple of 256 in order to achieve a strong uniformity property.

We describe the key generation here as a single algorithm, since the Uniform DH map does not follow the sample-and-encode paradigm. The algorithm is given in Figure 8.

The map relies on the fact that if x and y are even, $X = g^x \pmod p$, and $Y = g^y \pmod p$, then $(p - Y)^x = Y^x \pmod p$ and $(p - X)^y = X^y \pmod p$. The uniformity in this case depends on the size of the group. Uniform DH was originally proposed as a mapping for MODP groups 5 and 14 [39]. We note that group 5 was downgraded to SHOULD NOT in RFC 8247 [45], as the security margin was considered too narrow due to group size. For completeness, we describe the statistical distance from uniform when embedding keys in such a group anyway.

In group 5, we have $p_5 = 2^{1536} - 2^{1472} - 1 + 2^{64} \cdot (2^{1406}\pi + 741804)$. Let $\delta_5 = 2^{1536} - p_5 \approx 2^{1470}$. In group 14, we have $p_{14} = 2^{2048} - 2^{1984} - 1 + 2^{64} \cdot (2^{1918}\pi + 124476)$ and $\delta_{14} = 2^{2048} - p_{14} \approx 2^{1982}$. Let $\mathcal{U}_d$ be a uniform distribution over d bit integers (where $d = 1536$ or $d = 2048$), and let $\mathcal{Z}_k$ be the distribution of the respective Uniform DH map. The public key uniformity (cf. Definition 2.4) for the map over group k is then the statistical distance between distributions $\mathcal{Z}_k$ and $\mathcal{U}_d$:

$$\begin{aligned}\Delta(\mathcal{U}_d, \mathcal{Z}_k) &= \frac{1}{2} \sum_{\alpha \in [0, 2^d - 1]} \left| \Pr[\mathcal{Z}_k = \alpha] - \Pr[\mathcal{U}_d = \alpha] \right| \\ &= \frac{1}{2} \left((p_k - \delta_k) \cdot \left| \frac{1}{p_k} - \frac{1}{p_k + \delta_k} \right| + 2\delta_k \cdot \left| \frac{1}{2p_k} - \frac{1}{p_k + \delta_k} \right| \right).\end{aligned}$$

For both MODP group 5 and group 14, this is approximately 2^{-66}. Our analysis confirms the calculations from [64, Appendix A].

Any odd values of the private key x are rejected in the encoding function. Therefore, given that p is an odd prime, the first-sample success probability (cf. Definition 2.3) is $\frac{1}{2}$.

Telex *original-or-twist [67].* Telex uses a *tagging* mechanism that embeds information into a uniformly random string that is only usable to a party with the corresponding secret information. This tag functions similarly to the marker M_C in the `obfs4` protocol; however, the Telex tag also includes an elliptic curve public key. Here, we only describe the initial part of the tag which includes the public key.

```
UDH.KGen():
    $x \xleftarrow{\$} [0, p-2]$   // private key
    if $x \bmod 2 \neq 0$, return $\perp$
    $X \leftarrow g^x \pmod p$
    $b \xleftarrow{\$} \{0,1\}$
    $X_0 \leftarrow X$
    $X_1 \leftarrow p - X$
    return $x, X_b, X_b$
```

```
von Ahn–Hopper.KGen():
    $x \xleftarrow{\$} [0, p-2]$   // private key
    $X \leftarrow g^x \pmod p$
    if MSB of $X$ is 1, return $\perp$
    $X' \leftarrow$ all bits of $X$ except MSB
    return $x, X, X'$
```

```
Telex.KGen():
    $s \xleftarrow{\$} [1, p-1]$   // private key
    $b \xleftarrow{\$} \{0,1\}$
    $\beta \leftarrow s \cdot g_b$   // x-coordinate only
    return $s, \beta, \beta$
```

Figure 8: UDH**, von Ahn–Hopper, and** Telex KGen **algorithms.**

Telex selects several public parameters. The elliptic curve $E = \{(x,y) : y^2 = x^3 - 3x + b\}$ is defined over a field of prime order p, $\mathbb{F}_p$, where $p \equiv 3 \mod 4$. Let ℓ_p denote the bit length of p. The *twist* curve is then $E' = \{(x,y) : -y^2 = x^3 - 3x + b\}$. b must be chosen such that E and E' have prime order over $\mathbb{F}_p$. This ensures that for any $a \in \mathbb{F}_p$, a is either the x-coordinate of a point on E or it is the x-coordinate of a point on E'. The idea of using a curve or its twist for random ciphertexts was similarly proposed by Möller [44].

To setup the encoding, the following steps are taken:

- Two public hash functions, H_1 and H_2, are defined.
- The x-coordinates of generators g_0 and g_1 for curves E and E', respectively, are published as public information.
- A random private key $r \in \{0,1\}^{\ell_p}$ is selected. The x-coordinates of $\alpha_0 = r \cdot g_0$ and $\alpha_1 = r \cdot g_1$ are published. These must not be the point at infinity.

We describe the key generation here as a single algorithm, since the Telex map does not follow the sample-and-encode paradigm. The algorithm is given in Figure 8. The inverse operation is omitted since the public key is the same as its encoded variant.

The public key uniformity (cf. Definition 2.4) of this encoding is then the statistical distance between the output distribution of the encoding and the uniform distribution. That is,

$$\frac{1}{2} \left(p \cdot \left| \frac{1}{p} - \frac{1}{2^{\ell_p}} \right| + (2^{\ell_p} - p) \cdot \left| \frac{1}{2^{\ell_p}} \right| \right).$$

For the parameters used in the Telex implementation, $p = 2^{168} - 2^8 - 1$ and $\ell_p = 168$, this is approximately 2^{-160}. There is no rejection of values in the key generation algorithm, so the first-sample success probability is 1.

Elligator^2 [58]. To address some shortcomings of the Elligator2 encoding (in particular, the fact that only $\approx 1/2$ the points on a curve can be encoded and the restrictions on the types of Elliptic curves that can be encoded), Tibouchi proposed an alternative encoding for elliptic curve points, named Elligator^2. The construction first fixes a suitable, injective point encoding function which maps points from a finite field to an elliptic curve defined over the field, i.e., $f\colon GF(q) \rightarrow E(GF(q))$. Then, a point $P \in E(GF(q))$ is represented as a random preimage of the tensor square function $f^{\otimes 2}\colon (u,v) \in (f^{\otimes 2})^{-1}(P) \subset GF(q)^2$. Here, the tensor square function $f^{\otimes 2}\colon GF(q)^2 \longrightarrow E(GF(q))$ maps (u,v) to $f(u) + f(v)$. Elements of $GF(q)^2$ are then mapped directly to their bit-representation when

q is close to a power of 2. The encoding therefore transmits data corresponding to both u and v for a given point $P = (u, v)$. As a result, the output is approximately twice as long as an Elligator2 encoding, but has a first-sample success probability of 1 (as opposed to $\approx 1/2$). Additionally, the statistical distance from uniform is $2 \cdot (1 - q/2^n)$ where n is the bitlength of q. For example, for Curve25519 this value is $\approx 2^{-249.752}$.

The problem then reduces to finding a suitable *well-bounded* encoding function $f : GF(q) \rightarrow E(GF(q))$ whose inverse is efficiently computable. We refer the reader to [58] for appropriate definitions of well-boundedness and constructions of specific encoding functions. Several proposals are given for different classes of curves including ordinary curves where $p \equiv 3 \bmod 4$, Elligator 1-compatible curves, and Barreto–Naehrig curves [58, §4].

Alternative Kemeleon *ciphertext encoding.* We briefly describe a slightly modified alternative encoding for ML-KEM ciphertexts, than the one described in Section 2.4. In the current approach, we propose rejecting values of c_2 based on the coefficients that are equal to 0. An alternative approach would be to forgo the rejection of c_2 and instead perform the same decompression and randomness recovery as is done for c_1. In this method, the encoding algorithm Kemeleon.Encode is applied to the concatenated vector of coefficients from c_1 *and* c_2. Although c_1 is a vector of k polynomials and c_2 is a single polynomial, this can simply be treated as a vector of $(k+1) \cdot n$ coefficients. Then, the resulting first-encap success probability becomes $2^{\lceil \log_2(q^{n \cdot (k+1)}+1) \rceil - 1} / q^{n \cdot (k+1)}$, which is $\approx 2^{-0.694}$ for $k = 3$. This is strictly worse because the first-encap success probability decreases and the output size of encoded ciphertexts increases. On the other hand, for $k = 2$, this probability becomes $\approx 2^{-0.27}$ which is much better than the previous $2^{-0.957}$; however, this approach comes with the tradeoff of increased output size. In any case, the first-encap success probability is greater than 1/2 and the statistical distance from uniform remains as 0.

C Proof of Theorem 4.3

Assume $\mathcal{A}$ makes q_{T} queries to the Test oracle. Since $\mathcal{A}$ can test each session at most once (assuming w.l.o.g. that it does not make useless queries that definitely result in a $\perp$ response), we have that $q_{\mathrm{T}} \leq n_s$.

By standard advantage term rewriting, we have

$$\begin{aligned} \mathsf{Adv}^{\mathsf{ObfKE}}_{\mathsf{KE},\mathcal{S}}(\mathcal{A}) &= 2 \cdot \Pr\left[\mathrm{G}^{\mathsf{ObfKE}}_{\mathsf{KE},\mathcal{S}}(\mathcal{A}) \Rightarrow 1\right] - 1 \\ &= \Pr\left[\mathrm{G}^{\mathsf{ObfKE}}_{\mathsf{KE},\mathcal{S},1}(\mathcal{A}) \Rightarrow 1\right] - \Pr\left[\mathrm{G}^{\mathsf{ObfKE}}_{\mathsf{KE},\mathcal{S},0}(\mathcal{A}) \Rightarrow 1\right], \end{aligned}$$

where $\mathrm{G}^{\mathsf{ObfKE}}_{\mathsf{KE},\mathcal{S},b}$ denotes the game $\mathrm{G}^{\mathsf{ObfKE}}_{\mathsf{KE},\mathcal{S}}$ with a fixed challenge bit b.

We describe the ObfKE-1 case in detail first, and derive the ObfKE-1* one from it afterwards. Define hybrid games $\mathrm{G}^{\mathsf{ObfKE}_n}_{\mathsf{KE},\mathcal{S}}$ for $n \in [0..n_s + n_r \cdot q_{\mathrm{C}}]$ that work like $\mathrm{G}^{\mathsf{ObfKE}}_{\mathsf{KE},\mathcal{S},1}$, except with the following changes to the Test and ChallExec oracles:

- Test(u, i): Let π^i_u be the tth session created in the game. If $t \leq n$, return k_0, else return k_1.
  (That is, we let Test return the random key k_0 for the first n sessions, and the real key k_1 for the remaining sessions.)
- ChallExec(u, v): Let v be the rth server created in the game (via a call to NewUser), and the ChallExec query be the tth query to that server. If $(r-1) \cdot q_{\mathrm{C}} + t \leq n - n_s$, return $(trans_0, k_0)$, else return $(trans_1, k_1)$. That is, we let ChallExec return a simulated transcript and random key, $(trans_0, k_0)$, when called on the first t calls with the rth server and on server created before. For the remaining calls to the rth server and to all later-created servers, ChallExec returns the real transcript and key, $(trans_1, k_1)$. Also, for the first n_s hybrids, we return the real transcript and key.

We have that $\mathrm{G}^{\mathsf{ObfKE}_0}_{\mathsf{KE},\mathcal{S}}$ equals the "real" game $\mathrm{G}^{\mathsf{ObfKE}}_{\mathsf{KE},\mathcal{S},1}$ for $b = 1$ and $\mathrm{G}^{\mathsf{ObfKE}_{n_s+n_r \cdot q_{\mathrm{C}}}}_{\mathsf{KE},\mathcal{S}}$ equals the "random" game $\mathrm{G}^{\mathsf{ObfKE}}_{\mathsf{KE},\mathcal{S},0}$ for $b = 0$. We will now bound each hop from $\mathrm{G}^{\mathsf{ObfKE}_{n-1}}_{\mathsf{KE},\mathcal{S}}$ to $\mathrm{G}^{\mathsf{ObfKE}_n}_{\mathsf{KE},\mathcal{S}}$ by the advantage of a reduction $\mathcal{B}$ against the single-challenge selective game $\mathrm{G}^{\mathsf{ObfKE\text{-}1}}_{\mathsf{KE},\mathcal{S}}$.

The reduction samples $n \xleftarrow{\$} [1..n_s + n_r \cdot q_{\mathrm{C}}]$ at random and commits to attack type A = Test and session $s = n$ if $s \leq n_s$, and to attack type A = ChallExec and server $p = \lceil (n - n_s)/q_{\mathrm{C}} \rceil$ otherwise. It then simulates the hybrid game $\mathrm{G}^{\mathsf{ObfKE}_n}_{\mathsf{KE},\mathcal{S}}$ for $\mathcal{A}$ by relaying the queries of $\mathcal{A}$ to its own oracles, except for queries to the Test and ChallExec oracles, which $\mathcal{B}$ handles as follows:

- Test(u, i): Let π^i_u be the tth session created in the game.
  - If $t < n$, $\mathcal{B}$ samples and returns a random key $k_0 \xleftarrow{\$} \mathsf{KE.KS}$.
  - If $t = n$, $\mathcal{B}$ makes its single Test oracle query on (u, i).
  - If $t > n$, $\mathcal{B}$ calls RevSessionKey(u, i) and returns the resulting (real) session key.
    Note that such RevSessionKey queries cannot violate the freshness predicate in $\mathcal{B}$'s game unless $\mathcal{A}$ violates Fresh in its game, since Test queries (by $\mathcal{A}$) cannot be made on sessions partnered with tested or revealed sessions.

  Note that $t = n$ implies that $n \leq n_s$ and hence $t = n = s$, so $\mathcal{B}$ correctly committed to testing the sth session. Also, for $n \geq n_s$, $\mathcal{B}$ samples all answers at random and never queries Test.
- ChallExec(u, v): Let v be the rth server created in the game and the query be the tth ChallExec query to v. Define $t^* := n_s + (r-1) \cdot q_{\mathrm{C}} + t$.
  - If $t^* < n$, $\mathcal{B}$ runs the simulator $\mathcal{S}$ and returns the resulting simulated transcript $trans_0$ and a random key $k_0 \xleftarrow{\$} \mathsf{KE.KS}$.
  - If $t^* = n$, $\mathcal{B}$ makes its single ChallExec oracle query on (u, v).
    Note that in $\mathcal{B}$'s game, only chall_v will be set to true since $\mathcal{B}$ simulates all other queries to ChallExec itself. This can only help $\mathcal{B}$, since the ObfFresh predicate (independent of regular or strong obfuscation) in its game might be satisfied even when $\mathcal{A}$ violates its predicate.
  - Else (if $t^* > n$, and in particular if $n \leq n_s$), $\mathcal{B}$ uses its Send oracle (for some new indices $i > n_s$ unused by $\mathcal{A}$) repeatedly to simulate the full protocol run between π_u and π_v in the ChallExec oracle. It concatenates the resulting message into $trans_1$ and reveals the session key (via RevSessionKey) and returns that transcript and key.
    Note that the initiator's first message is immediately "consumed" by sending it to the responder, hence the set $\mathcal{F}_v$ is

unchanged after $\mathcal{B}$ completed its sequence of Send oracle calls. Furthermore, $\mathcal{B}$'s RevSessionKey queries do not violate Fresh as it does not test any session.

Note that $t^* = n$ implies that $p = \lceil (n - n_s)/q_{\mathrm{C}} \rceil = r$, so $\mathcal{B}$ correctly committed to the challenged server v being the pth created one.

Finally, $\mathcal{B}$ outputs $\mathcal{A}$'s bit guess b' as its own.

We have that $\mathcal{B}$ simulates either hybrid game $\mathrm{G}^{\mathsf{ObfKE}_{n-1}}_{\mathsf{KE},\mathcal{S}}$ or $\mathrm{G}^{\mathsf{ObfKE}_{n}}_{\mathsf{KE},\mathcal{S}}$, depending whether the challenge bit in $\mathcal{B}$'s game $\mathrm{G}^{\mathsf{ObfKE\text{-}1}}_{\mathsf{KE},\mathcal{S},b}$ is $b = 1$ or $b = 0$. To see this, observe that when testing the $(t = n)$-th session, the response is real (simulating hybrid $n - 1$) if $b = 1$ in $\mathcal{B}$'s game $\mathrm{G}^{\mathsf{ObfKE\text{-}1}}_{\mathsf{KE},\mathcal{S},b}$ and random (simulating hybrid n) if $b = 0$. Likewise, for the tth query to ChallExec for the pth server, the transcript and key are real if $b = 1$ and random if $b = 0$. We hence have

$$\begin{aligned}
&\Pr\left[\mathrm{G}^{\mathsf{ObfKE}_{n-1}}_{\mathsf{KE},\mathcal{S}}(\mathcal{A}) \Rightarrow 1\right] - \Pr\left[\mathrm{G}^{\mathsf{ObfKE}_{n}}_{\mathsf{KE},\mathcal{S}}(\mathcal{A}) \Rightarrow 1\right]\\
&\leq \Pr\left[\mathrm{G}^{\mathsf{ObfKE\text{-}1}}_{\mathsf{KE},\mathcal{S},1}(\mathcal{B}) \Rightarrow 1\right] - \Pr\left[\mathrm{G}^{\mathsf{ObfKE\text{-}1}}_{\mathsf{KE},\mathcal{S},0}(\mathcal{B}) \Rightarrow 1\right]\\
&= \mathsf{Adv}^{\mathsf{ObfKE\text{-}1}}_{\mathsf{KE},\mathcal{S}}(\mathcal{B}).
\end{aligned}$$

Summing over the hybrids $\mathrm{G}^{\mathsf{ObfKE}_{n}}_{\mathsf{KE},\mathcal{S}}$ for $n \in [1..n_s + n_r \cdot q_{\mathrm{C}}]$, and recalling that $\mathrm{G}^{\mathsf{ObfKE}}_{\mathsf{KE},\mathcal{S},1} = \mathrm{G}^{\mathsf{ObfKE}_{0}}_{\mathsf{KE},\mathcal{S}}$ and $\mathrm{G}^{\mathsf{ObfKE}}_{\mathsf{KE},\mathcal{S},0} = \mathrm{G}^{\mathsf{ObfKE}_{n_s+n_r \cdot q_{\mathrm{C}}}}_{\mathsf{KE},\mathcal{S}}$ yields the theorem bound.

ObfKE-1*. The case of ObfKE-1* works as above by setting $q_{\mathrm{C}} = 1$ and $t^* := n_s + r$, and having $\mathcal{B}$ relay all ChallExec queries (instead of just one) for the rth server to its own ChallExec oracle. □

D Security Proof for obfs4 (Theorem 5.1)

We proceed via a series of game hops.

Game 0. We start with the security game for regular obfuscation (rObfKE), $\mathrm{G}_0 = \mathrm{G}^{\mathsf{rObfKE}}_{\mathtt{obfs4},\mathcal{S}_{\mathtt{obfs4}}}$. So, omitting the $\mathcal{S}_{\mathtt{obfs4}}$ subscript from here on, we have

$$\mathsf{Adv}^{\mathsf{rObfKE}}_{\mathtt{obfs4}}(\mathcal{A}) = \mathsf{Adv}^{\mathrm{G}_0}_{\mathtt{obfs4}}(\mathcal{A}) = 2 \cdot \Pr[\mathrm{G}_0] - 1.$$

Game 1 (Exclude DH collisions). In Game G_1, we abort if two honest sessions sample the same DH share (X and Y) or if two servers sample the same long-term DH share B. By the birthday bound, across at most n_s honest sessions (for the former) and at most n_r servers (for the latter), the probability of such an abort can be upper-bounded by

$$\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_1] \leq \frac{n_s^2 + n_r^2}{q},$$

where q is the X25519 DH group order.

Establishing soundness (Sound)**.** We now observe that in Game G_1, the adversary cannot violate soundness anymore (i.e., the predicate Sound cannot be false). To see this, we check the conditions in Sound (Figure 5); recall that the session identifier is $\mathit{sid} := (X, Y, B, \mathit{NodeID})$ and the contributive identifier is $\mathit{cid} := (X)$.

(1) No triple sid match: For three session identifiers to match, two initiator sessions or two responder sessions would have to sample the same curve point. However, G_1 aborts in that case.
(2) Partnering implies different roles: For two same-role sessions to be partnered, they would have to sample the same curve point. Again, G_1 prevents that.
(3) Partnering implies same contributive identifiers: The contributive identifier contains a subset of entries in the session identifier. Hence, agreement on the latter implies agreement on the former.
(4) Partnering implies agreement on responder ID: The session identifier contains the involved server's long-term DH share B. As these do not collide by G_1, they uniquely identify the responder.
(5) Partnering implies same key: The elements in the session identifier uniquely determine the inputs to the computation of the session key *skey*, hence agreement on the former implies agreement on the latter.

Game 2 (Prevent Probed **being set).** In Game G_2, we stop setting Probed ← true (i.e., $\mathcal{A}$ cannot win anymore in G_2 by breaking probing resistance). We have

$$\Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_2] \leq \Pr[\mathrm{G}_1 \text{ sets Probed}]$$

and will bound the latter probability via the following, branching game hops $\mathrm{G}_{1.1}$ and $\mathrm{G}_{1.2}$.

Game 1.1 (Exclude *NodeID* collisions). In Game $\mathrm{G}_{1.1}$, we abort if two servers sample the same *NodeID* value in Setup. By a birthday bound over the n_r many servers and the 160-bit *NodeID* values, we have

$$\Pr[\mathrm{G}_1 \text{ sets Probed}] - \Pr[\mathrm{G}_{1.1} \text{ sets Probed}] \leq \frac{n_r^2}{2^{160}}.$$

Game 1.2 (Sample $\mathsf{HMAC}(\ldots \mathit{NodeID} \ldots)$ at random). In Game $\mathrm{G}_{1.2}$, we let the challenger sample the output of any HMAC call involving some *NodeID* input uniformly (but consistently) at random, instead of querying the HMAC random oracle, where the *NodeID* values are the random node IDs sampled for every server, i.e., every user u with $u.\mathit{role}$ = responder. However, from the moment RevPublicKey or RevSecretKey is queried on u (revealing the *NodeID* of u to $\mathcal{A}$), the challenger programs these values into the random oracle.

This change is observable for $\mathcal{A}$ only if it makes a random oracle query involving some user's *NodeID* prior to that *NodeID* being revealed. As *NodeID* occurs at three distinct, fixed places in HMAC inputs, as $\mathsf{HMAC}(B \| \mathit{NodeID}, \cdot)$, in *secret_input*, and in *auth_input*, each random oracle query might match with one of the three input types. Recalling that there are n_r many uniformly random *NodeID* values of length 160 bits each, we can upper bound the probability that $\mathcal{A}$ makes such a query by

$$\Pr[\mathrm{G}_{1.1} \text{ sets Probed}] - \Pr[\mathrm{G}_{1.2} \text{ sets Probed}] \leq \frac{3 q_{\mathrm{RO}} \cdot n_r}{2^{160}}.$$

Now observe that Probed in $\mathrm{G}_{1.2}$ is set if a non–initiator-first message m yields a reply by an unrevealed responder. Such a message m can be either (1) a replay of an already consumed message output by an honest initiator for this responder, or (2) one that is different from all honest initiators' first messages sent to this responder.

In the first case, m is rejected due to the replay check against the list of seen MAC values $st.\mathcal{S}_{MAC}$ recorded in the server's state.

In the second case, $m = X' \| P_C \| M_C \| MAC_C$ must be different from any initiator session's message sent to this responder. Since MAC_C is computed deterministically from $X' \| P_C \| M_C$ for a fixed responder and *NodeID* values do not collide by $\mathrm{G}_{1.1}$, it must be that no initiator session output $MAC_C = \mathsf{HMAC}(B \| NodeID, X' \| P_C \| M_C)$. By $\mathrm{G}_{1.2}$, adversary $\mathcal{A}$ also made no random oracle query matching $\mathsf{HMAC}(B \| NodeID, X' \| P_C \| M_C)$ as *NodeID* is unrevealed when m is received. Hence, the value $\mathsf{HMAC}(B \| NodeID, X' \| P_C \| M_C)$ computed by the unrevealed responder (via the random oracle) is an independent random 256-bit value when the Probed flag would be set in a SEND oracle call of $\mathrm{G}_{1.2}$. Across all q_{S} such calls, the adversary hence has the following guessing chance setting Probed in $\mathrm{G}_{1.2}$:

$$\Pr[\mathrm{G}_{1.2} \text{ sets Probed}] \leq \frac{q_{\mathrm{S}}}{2^{256}}.$$

This completes the branching, bounding $\Pr[\mathrm{G}_1 \text{ sets Probed}] \leq \frac{n_r^2}{2^{160}} + \frac{3q_{\mathrm{RO}} \cdot n_r}{2^{160}} + \frac{q_{\mathrm{S}}}{2^{256}}$. We now continue with the main proof from G_2, where Probed is never set.

Game 3 (Prevent ExplicitAuth **being violated).** In Game G_3, we abort the game if ExplicitAuth is violated (i.e., $\mathcal{A}$ cannot win anymore in G_3 by breaking explicit authentication). We have

$$\Pr[\mathrm{G}_2] - \Pr[\mathrm{G}_3] \leq \Pr[\neg\mathsf{ExplicitAuth} \text{ in } \mathrm{G}_2].$$

Recall that for ExplicitAuth to be violated, an initiator session must accept with a peer whose secret key is uncompromised at this point in the game, but for which there is no session of that peer holding the same session identifier. Formally,

$$\begin{aligned}&\Pr[\neg\mathsf{ExplicitAuth} \text{ in } \mathrm{G}_2]\\ &= \Pr\big[\exists \pi_u^i : u.role = \mathsf{initiator} \wedge \pi_u^i.\mathsf{t}_{\mathrm{acc}} < \mathsf{revsk}_{\pi_u^i.peerid} \wedge \\ &\qquad \forall \pi_v^j \text{ s.t. } v = \pi_u^i.peerid : \pi_u^i.sid \neq \pi_v^j.sid\big].\end{aligned}$$

We will bound this probability again through a series of branching game hops $\mathrm{G}_{2.1}$ and $\mathrm{G}_{2.3}$.

Game 2.1 (Guess violated initiator session and peer). In Game $\mathrm{G}_{2.1}$, we guess a "target session" π^*, the first session (in order of creation) which makes ExplicitAuth evaluate to false, as well as that session's peer, $v^* = \pi^*.peerid$. We let the game abort if that guess is incorrect, i.e., if ExplicitAuth is not violated when the session π^* accepts or if v^* is not its peer. This introduces a corresponding loss in the number of sessions and servers:

$$\Pr[\neg\mathsf{ExplicitAuth} \text{ in } \mathrm{G}_2] \leq n_s \cdot n_r \cdot \Pr[\neg\mathsf{ExplicitAuth} \text{ in } \mathrm{G}_{2.1}].$$

Game 2.2 (Stop resampling in KGen**).** Recall that the obfuscated ephemeral DH share generation $(x, X, X') \xleftarrow{\$} \mathsf{X2Ell2.KGen}()$ is of "sample-and-encode" type (cf. Definition 2.2). In Game $\mathrm{G}_{2.2}$, we abort if in the first session violating ExplicitAuth, π^*, the first DH share sampled in $\mathsf{X2Ell2.KGen}()$ *cannot* be successfully encoded (i.e., $\mathsf{Encode}(X) = \bot$). Game $\mathrm{G}_{2.2}$ equals Game $\mathrm{G}_{2.1}$ if no resampling is necessary in the latter, i.e., with probability equal the first-sample success probability of X2Ell2:

$$\Pr[\neg\mathsf{ExplicitAuth} \text{ in } \mathrm{G}_{2.1}] \leq 1/\epsilon_{\mathsf{X2Ell2}}^{\mathsf{1succ}} \cdot \Pr[\neg\mathsf{ExplicitAuth} \text{ in } \mathrm{G}_{2.2}].$$

Game 2.3 (GapDH in X, B). In Game $\mathrm{G}_{2.3}$, we replace the outputs of the random oracle HMAC using as input *secret_input* containing $\mathsf{DH}(X, B)$ with uniformly random values. This in particular means the *verify* value derived in π^* is replaced by some uniform value *verify**.

We can bound this game hop by a GapDH reduction $\mathcal{B}_1$ which embeds its GapDH challenge in the ephemeral DH share X of the target session π^* and its peer's long-term public key B. Note that by Game $\mathrm{G}_{2.2}$, X is generated as regular DH share without resampling. [7] Also, π^* violating ExplicitAuth guarantees that upon acceptance of π^*, b is unrevealed. We let $\mathcal{B}_1$ simulate sessions of v^* without knowledge of b by using the DDH oracle to ensure consistency of responses to HMAC random oracle and REVSESSIONKEY queries: whenever HMAC would need to be evaluated on an input involving Z^b for some DH share Z, $\mathcal{B}_1$ checks whether $\mathcal{A}$ made a corresponding random oracle query (identifiable via Z and B) with the potential DH secret C by querying $\mathrm{DDH}(Z, B, C)$.

Unless $\mathcal{A}$ makes a query involving the DH secret $\mathsf{DH}(X, B)$ prior to π^* accepting, it cannot detect the replacements upon input *secret_input* (including *verify**). If $\mathcal{A}$ makes such query, $\mathcal{B}_1$ is able to detect this (using its DDH oracle) and wins the GapDH game by outputting the CDH solution $\mathsf{DH}(X, B)$. Hence we have

$$\Pr[\neg\mathsf{ExplicitAuth} \text{ in } \mathrm{G}_{2.2}] - \Pr[\neg\mathsf{ExplicitAuth} \text{ in } \mathrm{G}_{2.3}] \leq \mathsf{Adv}_{\mathsf{X25519}}^{\mathsf{GapDH}}(\mathcal{B}_1).$$

Finally, in Game $\mathrm{G}_{2.3}$, the target session π^* computes the authentication value *auth* using HMAC on input a uniformly random value *verify*$^* \in \{0,1\}^{256}$ unknown to $\mathcal{A}$ prior to π^* accepting. Explicit authentication being violated for π^* means that no session of v^* can have computed this value *auth*, as its input include the session identifier components $sid = (X, Y, B, NodeID)$ and such session would otherwise be partnered with π^* (and hence ExplicitAuth satisfied). So, the only chance for making π^* accept without partner session is to correctly guess the 256-bit output *auth* of the random oracle HMAC on an unknown 256-bit input, the probability of which is

$$\Pr[\neg\mathsf{ExplicitAuth} \text{ in } \mathrm{G}_{2.3}] \leq \frac{1}{2^{256}}.$$

This completes the branching, with $\Pr[\neg\mathsf{ExplicitAuth} \text{ in } \mathrm{G}_2] \leq n_s \cdot n_r \cdot 1/\epsilon_{\mathsf{X2Ell2}}^{\mathsf{1succ}} \cdot \left(\mathsf{Adv}_{\mathsf{X25519}}^{\mathsf{GapDH}}(\mathcal{B}_1) + \frac{1}{2^{256}}\right)$. We now continue with the main proof from G_3, where ExplicitAuth is never violated.

Game 4 (Single-challenge selective security). We now restrict the adversary to a single-challenge selective (rObfKE-1*) version G_4 of game G_3, where $\mathcal{A}$ has to commit upfront to winning either

[7] Alternatively, we can forgo $\mathrm{G}_{2.2}$ and instead use the fact that GapDH challenges are rerandomizable (cf. [10, 17]): we would rerandomize the challenge used for X (where otherwise `obfs4` would sample a new DH value), undoing the randomization in the DDH queries. Rerandomization can be applied up to some limit, at which point the game would abort. This makes the subsequent loss (although still dependent on the first-sample success probability) arbitrarily small.

(I) via a TEST query on the pre-determined sth created session (A = TEST), or

(II) via (possibly multiple) CHALLEXEC queries on the pre-determined pth created server (A = CHALLEXEC).

Recall that in G_3, Sound is never violated, Probed never set, and ExplicitAuth never violated, hence $\mathcal{A}$ can only win via TEST or CHALLEXEC queries. Applying the hybrid argument from Theorem 4.3 (cf. Appendix C), which holds analogously for the so-modified Game G_3, we have

$$\mathsf{Adv}^{G_3}_{\mathsf{obfs4}}(\mathcal{A}) \leq (n_s + n_r) \cdot \mathsf{Adv}^{G_4}_{\mathsf{obfs4}}(\mathcal{A}).$$

Let us write G_4^{T} and G_4^{C} for the game G_4 restricted to the two winning options of $\mathcal{A}$ ($\mathrm{A} \in \{\textsc{Test}, \textsc{ChallExec}\}$). By the union bound,

$$\mathsf{Adv}^{G_4}_{\mathsf{obfs4}}(\mathcal{A}) \leq \mathsf{Adv}^{G_4^{\mathrm{T}}}_{\mathsf{obfs4}}(\mathcal{A}) + \mathsf{Adv}^{G_4^{\mathrm{C}}}_{\mathsf{obfs4}}(\mathcal{A}).$$

We will bound each term separately, through the following proof cases I and II.

Case I. Win via TEST In this proof case, $\mathcal{A}$ commits to winning via a single TEST query on the sth created session, which we call π^*. For the test session to satisfy Fresh, we must have that there is an honest partner—i.e., a session π_p^* holding the same *cid* or *sid* ("passive execution")—or that the responder peer is uncompromised upon acceptance of the initiator session π^* ("forward secrecy"). Note that since ExplicitAuth is ensured to hold (by Game G_3), the latter implies that there actually must be an honest partner π_p^* holding the same *sid*, so we can at this point focus on such passive executions.

Essentially following that case in the proof for the ntor protocol [31, Theorem 1], we will embed a GapDH challenge in the ephemeral DH shares of the test session π^* and its partner π_p^*, turning the test session's key into a uniformly random one.

Game I.0. This case begins with Game G_4 conditioned on A = TEST, where the test session has an honest (*sid*- or *cid*-)partner:

$$\mathsf{Adv}^{G_{\mathrm{I.0}}}_{\mathsf{obfs4}}(\mathcal{A}) = \mathsf{Adv}^{G_4^{\mathrm{T}}}_{\mathsf{obfs4}}(\mathcal{A}).$$

Game I.1 (Guess the partner session). In Game $G_{\mathrm{I.1}}$, we guess the session π_p^* that is the honest *sid* partner (if π^* is an initiator) resp. *cid* partner (if π^* is a responder) of π^*. Aborting if the guess is incorrect, this introduces a loss in the number of sessions:

$$\mathsf{Adv}^{G_{\mathrm{I.0}}}_{\mathsf{obfs4}}(\mathcal{A}) \leq n_s \cdot \mathsf{Adv}^{G_{\mathrm{I.1}}}_{\mathsf{obfs4}}(\mathcal{A}).$$

Game I.2 (Stop resampling in KGen). Next, we abort in $G_{\mathrm{I.2}}$ if the first DH shared sampled in X2Ell2.KGen() in the π^* or π_p^* session *cannot* be successfully encoded ($\mathsf{Encode}(X) = \bot$). Game $G_{\mathrm{I.2}}$ equals Game $G_{\mathrm{I.1}}$ if no resampling is necessary, i.e., with probability equal the first-sample success probability of X2Ell2 squared:

$$\mathsf{Adv}^{G_{\mathrm{I.1}}}_{\mathsf{obfs4}}(\mathcal{A}) \leq (1/\epsilon^{\mathsf{1succ}}_{\mathsf{X2Ell2}})^2 \cdot \mathsf{Adv}^{G_{\mathrm{I.2}}}_{\mathsf{obfs4}}(\mathcal{A}).$$

Game I.3 (GapDH in X, Y). In Game $G_{\mathrm{I.3}}$, we replace the outputs of the random oracle HMAC on input *secret_input* containing $\mathsf{DH}(X, Y)$ with uniformly random values, where X and Y are the ephemeral DH shares sent or received by π^*. This in particular replaces the session key *skey* derived in π^* by a uniform random value.

We can bound this game hop by a GapDH reduction $\mathcal{B}_2$ which embeds its GapDH challenge in X and Y (which by $G_{\mathrm{I.2}}$ are generated as regular DH shares without resampling). If π^* is a responder session, π_p^* might receive a different ephemeral DH share than Y, in which case $\mathcal{B}_2$ uses its DDH oracle to ensure consistency with the HMAC random oracle.

Unless $\mathcal{A}$ makes a query involving the DH secret $\mathsf{DH}(X, Y)$, it cannot detect the replacement, and if it does, $\mathcal{B}_2$ detects this and outputs the CDH solution. So,

$$\Pr[G_{\mathrm{I.2}}] - \Pr[G_{\mathrm{I.3}}] \leq \mathsf{Adv}^{\mathsf{GapDH}}_{\mathsf{X25519}}(\mathcal{B}_2).$$

Finally, in $G_{\mathrm{I.3}}$, the real and random session key output of the TEST oracle are equally distributed. Also, any non-partnered session keys are independent of π^*. Hence, $\mathcal{A}$ has no better chance than guessing the challenge bit b and $\mathsf{Adv}^{G_{\mathrm{I.3}}}_{\mathsf{obfs4}}(\mathcal{A}) = 0$.

Case II. Win via CHALLEXEC In this proof case, $\mathcal{A}$ commits to challenging only the pth created server (and not making any TEST queries). Let $NodeID_p$ denote the node ID sampled in key generation of the pth user with *role* = responder; since it is sampled uniformly at random, we can have the game sample it already at the outset. We can assume that $\mathcal{A}$ makes at least one CHALLEXEC call, as otherwise its advantage is 0, and hence never calls REVPUBLICKEY or REVSECRETKEY on p, as otherwise it would violate ObfFresh and lose.

Game II.0. This case begins with Game G_4 conditioned on A = CHALLEXEC:

$$\mathsf{Adv}^{G_{\mathrm{II.0}}}_{\mathsf{obfs4}}(\mathcal{A}) = \mathsf{Adv}^{G_4^{\mathrm{C}}}_{\mathsf{obfs4}}(\mathcal{A}).$$

Game II.1 (Sample $\mathsf{HMAC}(\ldots NodeID_p \ldots)$ at random). In Game $G_{\mathrm{II.1}}$, we let the challenger sample the output of any HMAC call involving $NodeID_p$ as input uniformly (but consistently) at random, instead of querying the HMAC random oracle. This in particular affects the derived session keys *skey* and authentication values *auth* in CHALLEXEC sessions with server p. Note that, by the condition of ObfFresh, $NodeID_p$ will never be compromised. This change is hence observable for $\mathcal{A}$ only if it makes a random oracle query involving $NodeID_p$. As *NodeID* occurs at three distinct, fixed places in HMAC inputs, as $\mathsf{HMAC}(B \| NodeID, \cdot)$, in *secret_input*, and in *auth_input*, each random oracle query might match with one of the three input types. So we can upper bound the probability of $\mathcal{A}$ making a query matchin *NodeID* by

$$\Pr[G_{\mathrm{II.0}}] - \Pr[G_{\mathrm{II.1}}] \leq \frac{3q_{\mathrm{RO}}}{2^{160}}.$$

Game II.2 (Drop DH secrets from *secret_input*). From *secret_input* values containing $NodeID_p$ (and which are hence never queried to the random oracle), we can remove the DH secrets $X^y \| X^b$ (resp. $Y^x \| B^x$) without losing uniqueness of random oracle queries, since the further inputs $B \| X \| Y$ uniquely determines them. This change is unobservable to $\mathcal{A}$, hence

$$\Pr[G_{\mathrm{II.1}}] = \Pr[G_{\mathrm{II.2}}].$$

Game II.3 (Rewrite sampling DH values). Instead of computing the ephemeral DH shares of sessions in the ChallExec oracle as $(x, X, X') \xleftarrow{\$} \mathsf{X2Ell2.KGen}()$ and $(y, Y, Y') \xleftarrow{\$} \mathsf{X2Ell2.KGen}()$, in this game we sample $X' \xleftarrow{\$} \{0,1\}^{\mathsf{ol}}$ and $Y' \xleftarrow{\$} \{0,1\}^{\mathsf{ol}}$, and then compute the DH shares as $X \leftarrow \mathsf{X2Ell2.Decode}(X')$ and $Y \leftarrow \mathsf{X2Ell2.Decode}(Y')$. Note that due to the change in Game $\mathrm{G}_{\mathrm{II}.2}$, the challenger does not use the DH secrets x and y anymore.

The replacement of each such DH share can be bounded by the pk-unif security of X2Ell2, letting the reduction $\mathcal{B}_3$ use the obtained obfuscated public key $\hat{pk}$ in place of X' resp. Y', and its decoding $\mathsf{Decode}(\hat{pk})$ in place of X resp. Y. Via a standard hybrid argument over the q_{C} many queries to ChallExec, each involving two obfuscated DH public keys, we obtain

$$\Pr[\mathrm{G}_{\mathrm{II}.2}] - \Pr[\mathrm{G}_{\mathrm{II}.3}] \leq 2q_{\mathrm{C}} \cdot \mathsf{Adv}^{\mathsf{pk\text{-}unif}}_{\mathsf{X2Ell2}}(\mathcal{B}_3).$$

At this point, responses to the ChallExec oracle are distributed independent of the challenge bit b. In both cases, the returned transcript consists of random strings of length corresponding to the protocol messages and variable padding, matching the output of the simulator $\mathcal{S}_{\mathsf{obfs4}}$ (Section 5): X', Y' are replaced by random strings (by $\mathrm{G}_{\mathrm{II}.3}$), P_C, P_S are random padding by definition, and M_C, MAC_C, $auth$, M_S, MAC_S as well as the session key $skey$ are all replaced by uniformly random sampled values (by $\mathrm{G}_{\mathrm{II}.1}$) as they would be computed with HMAC on input $NodeID_p$.

Hence $\mathcal{A}$ cannot win in this case anymore and we have

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{II}.3}}_{\mathsf{obfs4}}(\mathcal{A}) \leq 0.$$

Collecting the bounds yields the theorem statement. □

E Security Proof for st-obfs (Theorem 6.1)

Proof. The security proof for the most part proceeds identically to that for obfs4 (cf. Appendix D). Ensuring soundness, preventing probing attacks, and preventing violations of explicit authentication applies unchanged, since the involved protocol parts are either unmodified or, in the case of probing resistance computing MAC_C still involves the *NodeID* input to HMAC (modeled as random oracle).

We restrict the adversary to a single-challenge selective ObfKE-1 version of the game (rather than ObfKE-1* as in the obfs4 proof). Thus, the adversary $\mathcal{A}$ has to commit to winning either

(I) via a Test query on the pre-determined sth created session (A = Test), or
(II) via *a single* ChallExec query on the pre-determined pth created server (A = ChallExec).

This differs from the use of *Theorem* 4.3 for obfs4 in that the adversary not cannot win via making *multiple* ChallExec queries – rather, $\mathcal{A}$ can make only one ChallExec query. Therefore, applying Theorem 4.3 (cf. Appendix C), we have

$$\mathsf{Adv}^{\mathrm{G}_3}_{\mathsf{st\text{-}obfs}}(\mathcal{A}) \leq (n_s + n_r \cdot q_{\mathrm{C}}) \cdot \mathsf{Adv}^{\mathrm{G}_4}_{\mathsf{st\text{-}obfs}}(\mathcal{A}).$$

Handling Case I of $\mathcal{A}$ winning via a single Test query applies unchanged, since the session key $skey$ is computed as before. It is indeed only Case II, treating $\mathcal{A}$ winning via a ChallExec query to the pre-determined pth-created server which obviously requires a different treatment, since now we want to establish strong obfuscation (instead of regular obfuscation for obfs4). In particular, we cannot rely on that server's $NodeID_p$ as an unknown input to HMAC anymore (cf. Appendix D, Game $\mathrm{G}_{\mathrm{II}.1}$), since the server's public key (containing $NodeID_p$) may be revealed in the strong obfuscation case. Case II for st-obfs then proceeds as follows.

Case II. Win via ChallExec In this proof case, $\mathcal{A}$ commits to challenging only the pth created server, on which it never calls RevSecretKey (as otherwise, ObfFresh for strong obfuscation would be violated). Denote that servers long-term key pair as (b_p, B_p). We assume that $\mathcal{A}$ makes one ChallExec call, as otherwise its advantage is 0.

Game II.0. This case begins with Game G_4 (up to that point identical to the obfs4 proof, cf. Appendix D) conditioned on A = ChallExec:

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{II}.0}}_{\mathsf{st\text{-}obfs}}(\mathcal{A}) = \mathsf{Adv}^{\mathrm{G}_4^{\mathrm{C}}}_{\mathsf{st\text{-}obfs}}(\mathcal{A}).$$

Game II.1 (Stop resampling in KGen). We abort in $\mathrm{G}_{\mathrm{II}.1}$ if either of the first DH shared sampled in $\mathsf{X2Ell2.KGen}()$ in the two sessions (initiator and responder) created by ChallExec *cannot* be successfully encoded ($\mathsf{Encode}(X) = \bot$). Game $\mathrm{G}_{\mathrm{II}.1}$ equals Game $\mathrm{G}_{\mathrm{II}.0}$ if no resampling is necessary, i.e., with probability equal the first-sample success probability of X2Ell2 squared:

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{II}.0}}_{\mathsf{st\text{-}obfs}}(\mathcal{A}) \leq (1/\epsilon^{\mathsf{1succ}}_{\mathsf{X2Ell2}})^2 \cdot \mathsf{Adv}^{\mathrm{G}_{\mathrm{II}.1}}_{\mathsf{st\text{-}obfs}}(\mathcal{A}).$$

Game II.2 (GapDH in X, B). In Game $\mathrm{G}_{\mathrm{II}.2}$, we replace the outputs of the random oracle HMAC using as input $\mathsf{DH}(X, B_p)$ in the sessions created by ChallExec with uniformly random values. This in particular replaces the MAC tags M_C, MAC_C, M_S, MAC_S, and $auth$ as well as the session key $skey$ in these sessions with uniformly random values.

We can bound this game hop by a GapDH reduction $\mathcal{B}_3$ which embeds its GapDH challenge (U, V) as follows: it embeds V in the pth server's long-term public B_p. For the initiator session created by ChallExec, $\mathcal{B}_3$ embeds a randomized version U^r (for a fresh random r) of U in the client's ephemeral public key X (which by $\mathrm{G}_{\mathrm{II}.1}$ is generated as a regular DH share without resampling). We let $\mathcal{B}_3$ simulate other sessions of the pth-created server without knowledge of b by using the DDH oracle to ensure consistency of responses to HMAC random oracle and RevSessionKey queries. Recall that $\mathcal{B}_3$ never has to reveal the pth-created server's long-term secret key, as otherwise ObfFresh would be violated.

Unless $\mathcal{A}$ makes a query involving the DH secret $\mathsf{DH}(X, B_p)$ of the sessions created by ChallExec, it cannot detect the replacements made. If $\mathcal{A}$ makes such a query, $\mathcal{B}_3$ is able to detect this (using its DDH) and wins the GapDH game by outputting the CDH solution $\mathsf{DH}(X^{-r}, B)$, where r is the randomizer for the corresponding X value. Hence we have

$$\Pr[\mathrm{G}_{\mathrm{II}.1}] - \Pr[\mathrm{G}_{\mathrm{II}.2}] \leq \mathsf{Adv}^{\mathsf{GapDH}}_{\mathsf{X25519}}(\mathcal{B}_3).$$

Game II.3 (Rewrite sampling DH values). Now we proceed as in Game $\mathrm{G}_{\mathrm{II}.3}$ in the obfs4 proof. Instead of computing the ephemeral DH shares in the ChallExec sessions as $(x, X, X') \xleftarrow{\$} \mathsf{X2Ell2.KGen}()$ and $(y, Y, Y') \xleftarrow{\$} \mathsf{X2Ell2.KGen}()$, in Game $\mathrm{G}_{\mathrm{II}.3}$ we sample $X' \xleftarrow{\$} \{0,1\}^{\mathsf{ol}}$ and $Y' \xleftarrow{\$} \{0,1\}^{\mathsf{ol}}$, and then compute the DH shares as $X \leftarrow \mathsf{X2Ell2.Decode}(X')$ and $Y \leftarrow \mathsf{X2Ell2.Decode}(Y')$.

Note that due to Game $G_{II.2}$, the challenger does not use the DH secrets x and y anymore, as all HMAC computations where these are involved have been replaced with random sampling in the ChallExec sessions.

The replacement of each such DH share can be bounded by the pk-unif security of X2Ell2, letting the reduction $\mathcal{B}_4$ use the obtained obfuscated public key $\hat{pk}$ in place of X' resp. Y', and its decoding $\mathsf{Decode}(\hat{pk})$ in place of X resp. Y. As there are two obfuscated DH public keys in the two sessions created by ChallExec, we obtain

$$\Pr[G_{II.2}] - \Pr[G_{II.3}] \leq 2 \cdot \mathsf{Adv}^{\mathsf{pk\text{-}unif}}_{\mathsf{X2Ell2}}(\mathcal{B}_4).$$

At this point, the response to the ChallExec oracle is distributed independent of the challenge bit b: In both cases, the returned transcript consists of random strings of length corresponding to the protocol messages and variable padding, as output by the simulator $\mathcal{S}_{\mathtt{obfs4}}$ (cf. Section 5): X', Y' are replaced by random strings (by $G_{II.3}$), P_C, P_S are random padding by definition, and M_C, MAC_C, M_S, MAC_S, and $auth$ as well as the session key $skey$ are all replaced by uniformly random sampled values (by $G_{II.2}$).

Hence $\mathcal{A}$ cannot win in this case anymore and we have

$$\mathsf{Adv}^{G_{II.3}}_{\mathtt{st\text{-}obfs}}(\mathcal{A}) \leq 0.$$

Replacing the sum of the above advantage terms for Case II for those from the same proof case in `obfs4` yields the claimed overall bound. □

F Security Proof for `pq-obfs` (Theorem 7.1)

We proceed via a series of game hops.

Game 0. We start with the security game for strong obfuscation (sObfKE), $G_0 = G^{\mathsf{sObfKE}}_{\mathtt{pq\text{-}obfs},\mathcal{S}_{\mathtt{pq\text{-}obfs}}}$, omitting the $\mathcal{S}_{\mathtt{pq\text{-}obfs}}$ subscript from here on:

$$\mathsf{Adv}^{\mathsf{sObfKE}}_{\mathtt{pq\text{-}obfs}}(\mathcal{A}) = \mathsf{Adv}^{G_0}_{\mathtt{pq\text{-}obfs}}(\mathcal{A}) = 2 \cdot \Pr[G_0] - 1.$$

Game 1 (Exclude KEM public key collisions). We modify Game G_0 to abort if two honest sessions sample the same ephemeral KEM key (pk_e) or if two servers sample the same long-term KEM key (pk_S). This is bounded by the public key collision probability $\mathsf{pkcoll}_{\mathsf{OKEM}}$ of the obfuscated KEM for the at most n_s ephemeral and n_r long-term KEM keys (Definition 2.7):

$$\Pr[G_0] - \Pr[G_1] \leq \mathsf{pkcoll}_{\mathsf{OKEM}}(n_s + n_r).$$

Game 2 (KEM correctness). We modify Game G_1 to abort if for any keys $(sk, pk, \widehat{pk}) \xleftarrow{\$} \mathsf{KGen}()$ and encapsulation $(c, K, \widehat{c}) \xleftarrow{\$} \mathsf{Encap}(pk)$, we have that $K \neq \mathsf{Decap}(sk, \widehat{c})$. The probability of this happening for any tuple $(sk, pk, \widehat{pk}, c, \widehat{c})$ is upper-bounded by the correctness error δ_{OKEM} of the obfuscated KEM as defined in Definition 2.6. Since there are two such tuples per session, we can bound the probability of such an abort by

$$\Pr[G_1] - \Pr[G_2] \leq 2n_s \cdot \delta_{\mathsf{OKEM}}.$$

Establishing soundness (Sound**).** We observe that in Game G_2, the adversary cannot violate soundness anymore (i.e., the predicate Sound cannot be false). Checking the conditions in Sound (Figure 5), we have:

(1) No triple sid match: For three session identifiers to match, two initiator sessions or two responder sessios would have to sample the same KEM keys. However, G_1 aborts in that case.
(2) Partnering implies different roles: For two same-role sessions to be partnered, they would have to sample the same ephemeral KEM key. Again, G_1 prevents that.
(3) Partnering implies same contributive identifiers: The contributive identifier contains a subset of entries in the session identifier. Hence, agreement on the latter implies agreement on the former.
(4) Partnering implies agreement on responder ID: The session identifier contains the involved server's long-term KEM key pk_S. As these do not collide by G_1, they uniquely identify the responder.
(5) Partnering implies same key: For two partnered sessions to derive a different key, the inputs to the computation of $skey$ would have to differ. This may occur if the inputs to Encap (or Decap) are not consistent or if Encap (or Decap) has a correctness error. The latter case is prevented by G_2. For the former, the elements in the session identifier uniquely determine the inputs; note in particular that it uniquely identifies the responder (as per above) and hence the *NodeID* input. Hence agreement on session identifiers implies that the same key is generated.

Game 3 (Prevent Probed **being set).** In Game G_3, we stop setting Probed ← true (i.e., $\mathcal{A}$ cannot win anymore in G_3 by breaking probing resistance). We have

$$\Pr[G_2] - \Pr[G_3] \leq \Pr[G_2 \text{ sets Probed}].$$

Recall that for Probed to be set, a responder session, whose public key $(pk_S, NodeID)$ is not revealed, must reply to a non-initiator-first message m, with a non-empty message m'. We will bound this probability via the following, branching game hops $G_{2.1}$–$G_{2.3}$.

Game 2.1 (Guess violated server and session). In Game $G_{2.1}$, we guess a "target server" v^*, the first server (in order of creation) on which Probed is set to true, as well as the (first) session π^* in which this happens. We let the game abort if that guess is incorrect, i.e., if Probed is not set to true when the server responds in session π^* or if Probed is set to true on a response from a server that was created earlier. This introduces an according loss in the number of servers times the number of sessions:

$$\Pr[G_3 \text{ sets Probed}] \leq n_s \cdot n_r \cdot \Pr[G_{2.1} \text{ sets Probed}].$$

Game 2.2 (Random *ES***).** In Game $G_{2.2}$, we replace ES in any session with the target server v^* with uniformly random values ES^*.

We bound the difference in this step by a reduction to the PRF security of F_2. The reduction $\mathcal{B}_1$ does not sample *NodeID* for v^* itself, but instead uses its oracle to compute ES in sessions with v^* by calling its oracle on K_S. When the oracle output is real, reduction $\mathcal{B}_1$ exactly simulates $G_{2.1}$, whereas when the output is random, it simulates $G_{2.2}$. Therefore, because *NodeID* is unrevealed for server v^*,

$$\Pr[G_{2.1} \text{ sets Probed}] - \Pr[G_{2.2} \text{ sets Probed}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_2}(\mathcal{B}_1).$$

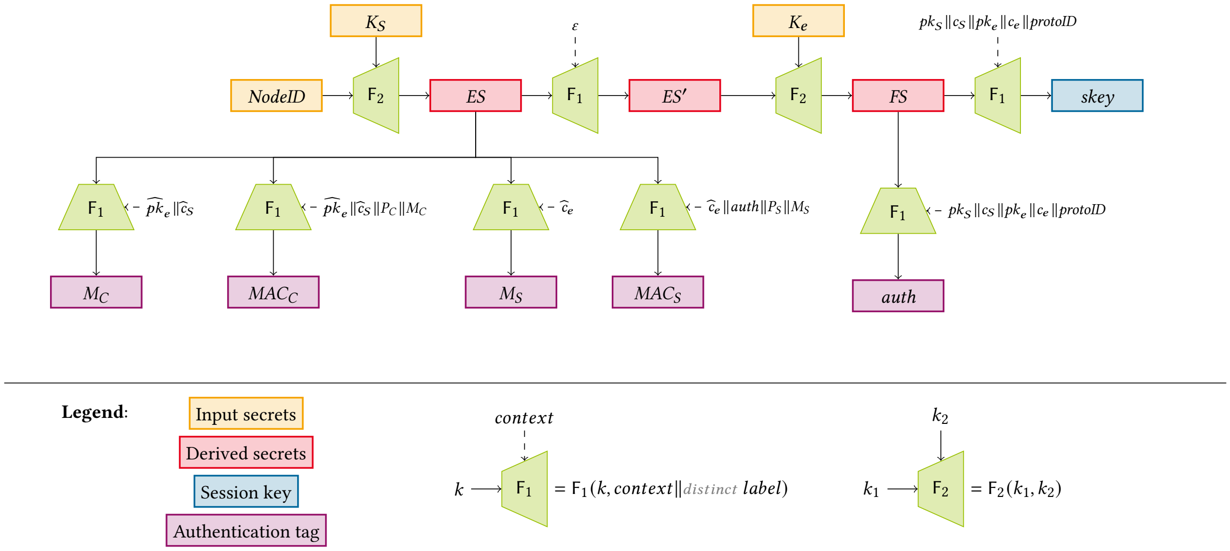

Figure 9: Key schedule for our post-quantum obfuscated key exchange protocol pq-obfs **(Figure 7).**

Game 2.3 (Random MAC_C**).** In Game $\mathsf{G}_{2.3}$, we replace any evaluation of F_1 on input random values ES^* in sessions with the target server v^* with a random function (per such value). This in particular replaces MAC_C in any session with v^* uniformly random values $MAC_C{}^*$.

We bound the difference in this step by a reduction to the PRF security of F_1. The reduction $\mathcal{B}_2$ does not sample ES^* in the target session π^* itself, but instead uses its oracle to compute $\mathsf{F}_1(ES^*, \cdot)$; in particular for computing MAC_C. Based on the oracle's output, $\mathcal{B}_2$ correctly simulates either $\mathsf{G}_{2.2}$ or $\mathsf{G}_{2.3}$. Hence,

$$\Pr[\mathsf{G}_{2.2} \text{ sets Probed}] - \Pr[\mathsf{G}_{2.3} \text{ sets Probed}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_2).$$

Observe that Probed in $\mathsf{G}_{2.3}$ is set if a non–initiator-first message m yields a reply by an unrevealed responder. Such a message m can be either (1) a replay of an already consumed message output by an honest initiator for this responder, or (2) one that is different from all honest initiators' first messages sent to this responder.

In the first case, m is rejected due to the replay check against the list of seen MAC values $st.\mathcal{S}_{MAC}$ recorded in the server's state.

In the second case, $m = \widehat{pk}_e \| \widehat{c}_S \| P_C \| M_C \| MAC_C$ must be different from any initiator session's message sent to this responder. By $\mathsf{G}_{2.3}$, the target client MAC value $MAC_C{}^*$ that the session π^* with v^* that first sets probed is a random fl_1-bit value unknown to $\mathcal{A}$. The adversary $\mathcal{A}$ can therefore only guess $MAC_C{}^*$ with probability

$$\Pr[\mathsf{G}_{2.3} \text{ sets Probed}] \leq \frac{1}{2^{\mathsf{fl}_1}}.$$

This completes the branching game hops bounding

$$\Pr[\mathsf{G}_2 \text{ sets Probed}] \leq n_s n_r \cdot \left(\mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_2}(\mathcal{B}_1) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_2) + \frac{1}{2^{\mathsf{fl}_1}}\right).$$

We now continue with the main proof from G_3, where Probed is never set.

Game 4 (Prevent ExplicitAuth **being violated).** In Game G_4, we abort the game if ExplicitAuth is violated (i.e., $\mathcal{A}$ cannot win anymore in G_4 by breaking explicit authentication). We have

$$\Pr[\mathsf{G}_3] - \Pr[\mathsf{G}_4] \leq \Pr[\neg\mathsf{ExplicitAuth} \text{ in } \mathsf{G}_3].$$

Recall that for ExplicitAuth to be violated, an initiator session must accept with a peer whose secret key is uncompromised at this point in the game, but for which there is no session of that peer holding the same session identifier. Formally,

$$\begin{aligned}&\Pr[\neg\mathsf{ExplicitAuth} \text{ in } \mathsf{G}_3]\\ &= \Pr\left[\exists \pi^i_u : u.role = \mathsf{initiator} \land \pi^i_u.\mathsf{t_{acc}} < \mathsf{revsk}_{\pi^i_u.peerid} \land\right.\\ &\qquad \left.\forall \pi^j_v \text{ s.t. } v = \pi^i_u.peerid : \pi^i_u.sid \neq \pi^j_v.sid\right].\end{aligned}$$

We will bound this probability again through a series of branching game hops $\mathsf{G}_{3.1}$–$\mathsf{G}_{3.6}$.

Game 3.1 (Guess violated initiator session and peer). In Game $\mathsf{G}_{3.1}$, we guess a "target session" π^*, the first session (in order of creation) which makes ExplicitAuth evaluate to false, as well as that session's peer, $v^* = \pi^*.peerid$. We let the game abort if that guess is incorrect, i.e., if ExplicitAuth is not violated when the session π^* accepts or if v^* is not its peer. This introduces an according loss in the number of sessions times the number of servers:

$$\Pr[\neg\mathsf{ExplicitAuth} \text{ in } \mathsf{G}_3] \leq n_s \cdot n_r \cdot \Pr[\neg\mathsf{ExplicitAuth} \text{ in } \mathsf{G}_{3.1}].$$

Game 3.2 (Long-term KEM IND-CCA**).** Let

$$(c_S, K_S, \widehat{c}_S) \xleftarrow{\$} \mathsf{OKEM.Encap}(pk_S)$$

be the encapsulation computed at session π^* with the long-term public key of v^*. In Game $\mathsf{G}_{3.2}$, we replace the long-term KEM key K_S with a uniformly random $\widetilde{K}_S$ in π^*. All values derived from K_S in π^* use the randomized value $\widetilde{K}_S$.

We bound the adversary $\mathcal{A}$'s difference in advantage by a reduction $\mathcal{B}_3$ to the IND-CCA security of OKEM.[8] $\mathcal{B}_3$ obtains the IND-CCA challenge $(pk, c^*, K^*, \widehat{c^*})$ and simulates the game for $\mathcal{A}$ as follows. It uses pk as the long-term public key of v^*. In π^*, $\mathcal{B}_3$ uses c^* as the ciphertext c_S and its encoding $\widehat{c^*}$ as $\widehat{c}_S$. In any session of v^*, if the ciphertext $\widehat{c}_S$ received is not $\widehat{c^*}$, then $\mathcal{B}_3$ queries its IND-CCA decapsulation oracle and uses the response as K_S; else, if $\widehat{c}_S = \widehat{c^*}$, then $\mathcal{B}_3$ uses K^* as K_S. Note that by the definition of ExplicitAuth, sk_S is not revealed to the adversary prior to π^* violating ExplicitAuth and thus $\mathcal{B}_3$ does not need to answer a RevSecretKey call on v^*. If K^* is the real KEM key then $\mathcal{B}_3$ has exactly simulated $\mathrm{G}_{3.1}$ to $\mathcal{A}$; else, if K^* is random, then $\mathcal{B}_3$ has exactly simulated $\mathrm{G}_{3.2}$ to $\mathcal{A}$. Therefore:

$$\begin{aligned}&\Pr[\neg\mathsf{ExplicitAuth}\text{ in }\mathrm{G}_{3.1}] - \Pr[\neg\mathsf{ExplicitAuth}\text{ in }\mathrm{G}_{3.2}]\\ &\leq \mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{OKEM}}(\mathcal{B}_3).\end{aligned}$$

Game 3.3 (Random *ES*). In Game $\mathrm{G}_{3.3}$, we replace ES with a uniformly random ES^* in π^*. We bound the difference in this step by a reduction to the swap-PRF security of F_2. The reduction $\mathcal{B}_4$ instead of sampling $\widetilde{K}_S$ uses its oracle to compute $\mathsf{F}_2(\cdot, \widetilde{K}_S)$ in π^*. When the output is real, reduction $\mathcal{B}_4$ exactly simulates $\mathrm{G}_{3.2}$, whereas when the output is random, it simulates $\mathrm{G}_{3.3}$. Therefore,

$$\begin{aligned}&\Pr[\neg\mathsf{ExplicitAuth}\text{ in }\mathrm{G}_{3.2}] - \Pr[\neg\mathsf{ExplicitAuth}\text{ in }\mathrm{G}_{3.3}]\\ &\leq \mathsf{Adv}^{\mathsf{swap\text{-}PRF}}_{\mathsf{F}_2}(\mathcal{B}_4).\end{aligned}$$

Game 3.4 (Random *ES′*). In Game $\mathrm{G}_{3.4}$, we replace evaluations $\mathsf{F}_1(ES^*, \cdot)$ in π^* with a random function. This in particular replaces ES' with a random value ES'^*. We bound the difference in this step with the PRF security of F_1, via a reduction $\mathcal{B}_5$ using its oracle in place of $\mathsf{F}_1(ES^*, \cdot)$:

$$\Pr[\neg\mathsf{ExplicitAuth}\text{ in }\mathrm{G}_{3.3}] - \Pr[\neg\mathsf{ExplicitAuth}\text{ in }\mathrm{G}_{3.4}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_5).$$

Game 3.5 (Random *FS*). In Game $\mathrm{G}_{3.5}$, we replace evaluations $\mathsf{F}_2(ES'^*, \cdot)$ in π^* with a random function. This in particular replaces FS with a random value FS^*. As in the previous hops, we can bound the difference by PRF security of F_2:

$$\Pr[\neg\mathsf{ExplicitAuth}\text{ in }\mathrm{G}_{3.4}] - \Pr[\neg\mathsf{ExplicitAuth}\text{ in }\mathrm{G}_{3.5}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_2}(\mathcal{B}_6).$$

Game 3.6 (Random *auth*). Finally, in Game $\mathrm{G}_{3.6}$, we replace evaluations $\mathsf{F}_1(FS^*, \cdot)$ in π^* with a random function. This in particular replaces $auth$ with a random value $auth^*$, and again can be bounded by PRF security of F_1:

$$\Pr[\neg\mathsf{ExplicitAuth}\text{ in }\mathrm{G}_{3.5}] - \Pr[\neg\mathsf{ExplicitAuth}\text{ in }\mathrm{G}_{3.6}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_7).$$

Now, given that $auth^*$ is a uniformly random value unknown to $\mathcal{A}$, violating ExplicitAuth in π^* requires that $\mathcal{A}$ correctly guesses the fl_1-bit value of $auth^*$. The probability of $\mathcal{A}$ guessing correctly is

$$\Pr[\neg\mathsf{ExplicitAuth}\text{ in }\mathrm{G}_{3.6}] \leq \frac{1}{2^{\mathsf{fl}_1}}.$$

This completes the branching, bounding

$$\begin{aligned}&\Pr[\neg\mathsf{ExplicitAuth}\text{ in }\mathrm{G}_3] \leq\\ &n_s n_r \cdot \Bigg(\mathsf{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{OKEM}}(\mathcal{B}_3) + \mathsf{Adv}^{\mathsf{swap\text{-}PRF}}_{\mathsf{F}_2}(\mathcal{B}_4) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_2}(\mathcal{B}_5)\\ &\qquad + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_6) + \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_7) + \frac{1}{2^{\mathsf{fl}_1}}\Bigg).\end{aligned}$$

We now continue with the main proof from Game G_4, where ExplicitAuth is never violated.

Game 5 (Single-challenge selective security). We now restrict the adversary to a single-challenge selective (sObfKE-1) version G_5 of Game G_4, where $\mathcal{A}$ has to commit upfront to winning either

(I) via a single Test query (A = Test) on the pre-determined s-th created session, or

(II) via a single ChallExec query (A = ChallExec) on the pre-determined p-th created server.

Recall that in G_4, Sound is never violated, Probed is never set, and ExplicitAuth is never violated, hence $\mathcal{A}$ can only win via Test or ChallExec queries. Applying the hybrid argument from Theorem 4.3 (which holds analogously for the so-modified Game G_4) we have

$$\mathsf{Adv}^{\mathrm{G}_4}_{\mathsf{pq\text{-}obfs}}(\mathcal{A}) \leq (n_s + n_r \cdot q_{\mathrm{C}}) \cdot \mathsf{Adv}^{\mathrm{G}_5}_{\mathsf{pq\text{-}obfs}}(\mathcal{A}).$$

Let us write $\mathrm{G}_5^{\mathrm{T}}$ and $\mathrm{G}_5^{\mathrm{C}}$ for the game G_5 restricted to the two winning options of $\mathcal{A}$ (A ∈ {Test, ChallExec}). By the union bound,

$$\mathsf{Adv}^{\mathrm{G}_5}_{\mathsf{pq\text{-}obfs}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathrm{G}_5^{\mathrm{T}}}_{\mathsf{pq\text{-}obfs}}(\mathcal{A}) + \mathsf{Adv}^{\mathrm{G}_5^{\mathrm{C}}}_{\mathsf{pq\text{-}obfs}}(\mathcal{A}).$$

We will bound each term separately, through the following proof cases I and II.

Case I. Win via Test In this proof case, $\mathcal{A}$ commits on winning via a single Test query on a session π^*. For the test session to satisfy Fresh, we must have that there is an honest partner—i.e., a session π_p^* holding the same *cid* or *sid* ("passive execution")—or that the responder peer is uncompromised upon acceptance of the initiator session π^* ("forward secrecy"). Note that since ExplicitAuth is ensured to hold (by Game G_4), the latter implies that there actually must be an honest partner π_p^* holding the same *sid*, so we can at this point focus on such passive executions.

Game I.0. This case begins with Game G_5 conditioned on A = Test, where the test session has an honest (*sid* or *cid*) partner:

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{I}.0}}_{\mathsf{pq\text{-}obfs}}(\mathcal{A}) = \mathsf{Adv}^{\mathrm{G}_5^{\mathrm{T}}}_{\mathsf{pq\text{-}obfs}}(\mathcal{A}).$$

Game I.1 (Guess the partner session). In Game $\mathrm{G}_{\mathrm{I}.1}$, we guess the session π_p^* that is the honest *sid* partner (if π^* is an initiator) or *cid* partner (else) of π^*. Aborting if the guess is incorrect, this introduces a loss in the number of sessions:

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{I}.0}}_{\mathsf{pq\text{-}obfs}}(\mathcal{A}) = n_s \cdot \mathsf{Adv}^{\mathrm{G}_{\mathrm{I}.1}}_{\mathsf{pq\text{-}obfs}}(\mathcal{A}).$$

Game I.2 (Ephemeral KEM IND-1CCA). Let

$$(c_e, K_e, \widehat{c}_e) \xleftarrow{\$} \mathsf{OKEM.Encap}(pk_e)$$

[8] Recall that for a sample/encaps-and-encode OKEM, IND-CCA is implied by the underlying KEM's IND-CCA security modulo the first-success probability of OKEM (Theorem 2.12). This applies in particular to our obfuscated ML-Kemeleon.

be the encapsulation computed at session π^*. If π^* is an initiator then this is the ephemeral public key pk_e in *sid*, and otherwise it is the ephemeral public key in *cid* (which must necessarily be determined by Game $\mathrm{G}_{\mathrm{I}.1}$).

In Game $\mathrm{G}_{\mathrm{I}.2}$, we replace the ephemeral KEM key K_e with a uniformly random $\widetilde{K}_e$ in π^*. All values derived from K_e in π^* use the randomized value $\widetilde{K}_e$.

We bound the adversary $\mathcal{A}$'s difference in advantage by a reduction $\mathcal{B}_8$ to the $\mathsf{IND\text{-}1CCA}$ security of OKEM. $\mathcal{B}_8$ obtains the $\mathsf{IND\text{-}1CCA}$ challenge $(pk, c^*, K^*, \widehat{c^*})$ and simulates the game for $\mathcal{A}$ as follows. In the protocol run between π^* and π^*_p, $\mathcal{B}_8$ uses pk as the ephemeral public key of the initiator and c^* as the ciphertext c_e and its encoding $\widehat{c^*}$ as the obfuscated ciphertext $\widehat{c}_e$ of the responder. If the initiator session receives a ciphertext $\widehat{c}_e \neq \widehat{c^*}$, then $\mathcal{B}_8$ queries its $\mathsf{IND\text{-}1CCA}$ decapsulation oracle (once) and uses the response as K_e; else, if $\widehat{c}_e = \widehat{c^*}$, then $\mathcal{B}_8$ uses K^* as K_e. If K^* is the real KEM key then $\mathcal{B}_8$ has exactly simulated $\mathrm{G}_{\mathrm{I}.1}$ to $\mathcal{A}$; else, if K^* is random, then $\mathcal{B}_8$ has exactly simulated $\mathrm{G}_{\mathrm{I}.2}$ to $\mathcal{A}$. Therefore:

$$\Pr[\mathrm{G}_{\mathrm{I}.1}] - \Pr[\mathrm{G}_{\mathrm{I}.2}] \leq \mathsf{Adv}^{\mathsf{IND\text{-}1CCA}}_{\mathsf{OKEM}}(\mathcal{B}_8).$$

Game I.3 (Random *FS*). In Game $\mathrm{G}_{\mathrm{I}.3}$, we replace evaluations $\mathsf{F}_2(\cdot, K^*)$ with a random function. This in particular replaces *FS* with a uniformly random value FS^* in π^*. (Note that if π^*_p received the same ciphertext, then it will also use K^*.)

We bound the difference in this step by a reduction to the swap-PRF security of F_2. The reduction $\mathcal{B}_9$ uses its oracle in place of $\mathsf{F}_2(\cdot, K^*)$, simulating either $\mathrm{G}_{\mathrm{I}.2}$ or $\mathrm{G}_{\mathrm{I}.3}$, giving:

$$\Pr[\mathrm{G}_{\mathrm{I}.2}] - \Pr[\mathrm{G}_{\mathrm{I}.3}] \leq \mathsf{Adv}^{\mathsf{swap\text{-}PRF}}_{\mathsf{F}_2}(\mathcal{B}_9).$$

Game I.4 (Random *skey*). Finally, in Game $\mathrm{G}_{\mathrm{I}.4}$, we replace evaluations $\mathsf{F}_1(FS^*, \cdot)$ with a random function. This in particular replaces *skey* with a uniformly random value $skey^*$ in π^*. We again bound this game hop by a reduction to the PRF security of F_1:

$$\Pr[\mathrm{G}_{\mathrm{I}.3}] - \Pr[\mathrm{G}_{\mathrm{I}.4}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_{10}).$$

We now have in Game $\mathrm{G}_{\mathrm{I}.4}$ that the real and random session key output of the $\textsc{Test}$ oracle are both randomly sampled. Also, any non-partnered session keys are independent of π^*, since the context input when deriving *skey* is precisely the session identifier *sid*. Hence, $\mathcal{A}$ has no better chance than guessing the challenge bit b and so

$$\mathsf{Adv}^{\mathrm{G}_{\mathrm{I}.4}}_{\mathsf{pq\text{-}obfs}}(\mathcal{A}) = 0.$$

Case II. Win via $\textsc{ChallExec}$ In this proof case, $\mathcal{A}$ commits on making a single $\textsc{ChallExec}$ query on the pth-created server, denoted v^* here (and not making any $\textsc{Test}$ queries). We assume that $\mathcal{A}$ makes one $\textsc{ChallExec}$ call, as otherwise its advantage is 0, and hence never calls $\textsc{RevSecretKey}$ on p, as otherwise it would violate ObfFresh and lose.

Game II.0. This case begins with Game G_5 conditioned on $\mathrm{A} = \textsc{ChallExec}$:

$$\Pr[\mathrm{G}_{\mathrm{II}.0}] = \Pr[\mathrm{G}^{\mathrm{C}}_5].$$

Game II.1 (Long-term KEM $\mathsf{SPR\text{-}CCA}$**).** Let

$$(c_S, K_S, \widehat{c}_S) \xleftarrow{\$} \mathsf{OKEM.Encap}(pk_S)$$

be the encapsulation computed by the initiator when $\textsc{ChallExec}$ is called.

In Game $\mathrm{G}_{\mathrm{II}.1}$, we replace the ciphertext c_S and its encoding $\widehat{c}_S$, as well as the long-term KEM key K_S as follows. Instead of running $\mathsf{OKEM.Encap}$, the initiator samples $\widehat{c}_S \xleftarrow{\$} \{0,1\}^{\mathsf{cl}}$ and $K_S \xleftarrow{\$} \mathcal{K}$ uniformly at random and derives the ciphertext as $c_S \leftarrow \mathsf{DecodeCtxt}(\widehat{c}_S)$.

We bound the adversary $\mathcal{A}$'s difference in advantage by a reduction $\mathcal{B}_{11}$ to the $\mathsf{SPR\text{-}CCA}$ security of OKEM.[9] $\mathcal{B}_{11}$ obtains the $\mathsf{SPR\text{-}CCA}$ challenge $(pk, c^*, K^*, \widehat{c^*})$ and simulates the game for $\mathcal{A}$ as follows. It uses pk as the public key of the pth-created responder v^* committed to by the adversary. When $\textsc{ChallExec}$ is called, $\mathcal{B}_{11}$ uses c^* as the ciphertext c_S and its encoding $\widehat{c^*}$ as $\widehat{c}_S$. In any session of v^*, if the ciphertext $\widehat{c}_S$ received is not $\widehat{c^*}$, then $\mathcal{B}_{11}$ queries its $\mathsf{SPR\text{-}CCA}$ decapsulation oracle and uses the response as K_S; else, if $\widehat{c}_S = \widehat{c^*}$, then $\mathcal{B}_{11}$ uses K^* as K_S. If $(c^*, K^*, \widehat{c^*})$ are the real KEM values then $\mathcal{B}_{11}$ exactly simulates $\mathrm{G}_{\mathrm{II}.0}$ to $\mathcal{A}$; else, if they are random, then $\mathcal{B}_{11}$ simulates $\mathrm{G}_{\mathrm{II}.1}$. Therefore:

$$\Pr[\mathrm{G}_{\mathrm{II}.0}] - \Pr[\mathrm{G}_{\mathrm{II}.1}] \leq \mathsf{Adv}^{\mathsf{SPR\text{-}CCA}}_{\mathsf{OKEM}}(\mathcal{B}_{11}).$$

Game II.2 (Random *ES*). In Game $\mathrm{G}_{\mathrm{II}.2}$, we replace *ES* with a uniformly random ES^* in the sessions created by the $\textsc{ChallExec}$ call. We bound the difference in this step by a reduction $\mathcal{B}_{12}$ to the swap-PRF security of F_2, using its oracle instead of computing $\mathsf{F}_2(\cdot, K_S)$. This yields

$$\Pr[\mathrm{G}_{\mathrm{II}.1}] - \Pr[\mathrm{G}_{\mathrm{II}.2}] \leq \mathsf{Adv}^{\mathsf{swap\text{-}PRF}}_{\mathsf{F}_2}(\mathcal{B}_{12})$$

Game II.3 (Random *ES′* and MAC tags). In Game $\mathrm{G}_{\mathrm{II}.3}$, we replace the evaluation $\mathsf{F}_1(ES^*, \cdot)$ with a random function in the sessions created by $\textsc{ChallExec}$. This replaces ES' and the MAC tags MAC_C, M_C, MAC_S, and M_S with uniformly random values ES'^*, $MAC_C{}^*$, $M_C{}^*$, $MAC_S{}^*$, and $M_S{}^*$, respectively. We again bound the difference in this by a reduction to the PRF security of F_1:

$$\Pr[\mathrm{G}_{\mathrm{II}.2}] - \Pr[\mathrm{G}_{\mathrm{II}.3}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_{13}).$$

Game II.4 (Random *FS*). In Game $\mathrm{G}_{\mathrm{II}.4}$, we replace *FS* with a uniformly random FS^* in the sessions created by the $\textsc{ChallExec}$ call, again bounded by PRF security of F_2:

$$\Pr[\mathrm{G}_{\mathrm{II}.3}] - \Pr[\mathrm{G}_{\mathrm{II}.4}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_2}(\mathcal{B}_{14}).$$

Game II.5 (Random *skey* and *auth*). In Game $\mathrm{G}_{\mathrm{II}.5}$, we replace evaluations $\mathsf{F}_1(FS^*, \cdot)$ with a random function in the sessions created by the $\textsc{ChallExec}$ call. This in particular replaces *skey* and *auth* with uniformly random values $skey^*$ and $auth^*$, respectively. Bounding again by PRF security of F_1:

$$\Pr[\mathrm{G}_{\mathrm{II}.4}] - \Pr[\mathrm{G}_{\mathrm{II}.5}] \leq \mathsf{Adv}^{\mathsf{PRF}}_{\mathsf{F}_1}(\mathcal{B}_{15}).$$

[9] Recall that for a sample/encaps-and-encode OKEM, SPR-CCA is implied by the underlying KEM's SPR-CCA security for "unform-encapsulation" simulators, modulo the first-success probability and ciphertext uniformity of OKEM (Theorem 2.13). This applies in particular to our obfuscated ML-Kemeleon.

Game II.6 (Random $\widehat{c}_e$). We finally replace the ephemeral public key and ciphertext in the transcript. In Game $G_{\mathrm{II}.6}$, we first replace the ephemeral obfuscated ciphertext $\widehat{c}_e$ with a uniform string of same length cl. This replacement can be bounded by the ciphertext uniformity (ctxt-unif) of OKEM, where the reduction $\mathcal{B}_{16}$ uses the obtained challenge ciphertext in place of $\widehat{c}_e$. Note since *FS* is sampled at random as per Game $G_{\mathrm{II}.4}$, $\mathcal{B}_{16}$ does not need to know K_e anymore. We obtain

$$\Pr[G_{\mathrm{II}.5}] - \Pr[G_{\mathrm{II}.6}] \leq \mathsf{Adv}^{\text{ctxt-unif}}_{\mathsf{OKEM}}(\mathcal{B}_{16}).$$

Game II.7 (Random $\widehat{pk}_e$). As the last step, we can in Game $G_{\mathrm{II}.7}$ replace the obfuscated ephemeral key $\widehat{pk}_e$ with a random string of same length ol. This can be bounded by the public key uniformity (pk-unif) of OKEM, having the reduction $\mathcal{B}_{17}$ use the obtained obfuscated public key in place of $\widehat{pk}_e$. We obtain

$$\Pr[G_{\mathrm{II}.6}] - \Pr[G_{\mathrm{II}.7}] \leq \mathsf{Adv}^{\text{pk-unif}}_{\mathsf{OKEM}}(\mathcal{B}_{17}).$$

Now we have that the response to the ChallExec oracle is distributed independent of the challenge bit b: In both cases, the returned transcript consists of random strings of length corresponding to the protocol messages and variable padding, exactly matching the output of the simulator $\mathcal{S}_{\texttt{pq-obfs}}$ (Section 7.2): $\widehat{pk}_e$ is random by $G_{\mathrm{II}.7}$, $\widehat{c}_S$ by $G_{\mathrm{II}.1}$, $\widehat{c}_e$ by $G_{\mathrm{II}.6}$, P_C, P_S are random padding by definition, and M_C, MAC_C, M_S, MAC_S are replaced by uniformly random values (by $G_{\mathrm{II}.3}$), as well as the *auth* tag and the session key *skey* (by $G_{\mathrm{II}.5}$).

Hence $\mathcal{A}$ cannot win in this case anymore and we have

$$\mathsf{Adv}^{G_{\mathrm{II}.7}}_{\texttt{pq-obfs}}(\mathcal{A}) = 0.$$

Collecting the bounds yields the theorem statement. □