

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.

CCS Concepts

• Security and privacy → Cryptography; Network security.

Keywords

Fully encrypted protocols, obfuscation, key exchange, obfs4, ML-KEM, quantum-safe

ACM Reference Format:

Felix Günther, Douglas Stebila, and Shannon Veitch. 2024. Obfuscated Key Exchange. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS ’24)*, October 14–18, 2024, Salt Lake City, UT, USA. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3658644.3690220>

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS ’24, October 14–18, 2024, Salt Lake City, UT, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0636-3/24/10

<https://doi.org/10.1145/3658644.3690220>

protocols. To evade this class of blocking, censorship circumvention practitioners have developed fully encrypted protocols (FEPs) like obfsproxy [45], 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 [40], 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, 39] 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 [42].

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 [20] 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 to establishing hidden connections, and yet no formal security definition for the handshake phase exists.

obfs2..3..4: an iterative design process. obfsproxy [43–45] 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 [46]) 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 [19]. 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 formalize obfuscated key exchange, capturing these goals concretely and, for the first time, providing tangible security assurance for the latest obfsproxy design obfs4 as an obfuscated

key exchange. 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 [51]. At the same time, whitelisting is generally disincentivized for censors due to high collateral damage [7, 47] and so the strategy of “avoiding blacklists” by looking random is effective in practice. Therefore, we take a more general approach and 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 distribution of messages in the protocol, allowing our definitions to capture a wide range of protocol classes including “looking like random” or matching a whitelisted protocol such as TLS. Concretely, the simulator-based definition can 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 7.2). 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 [33] Diffie–Hellman (DH) public keys), Uniform DH [26], and Telex’s original-or-twist method [52].

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

Property	obfs family [43–45]			Our protocols		Our model
	obfs2	obfs3	obfs4	st-obfs	pq-obfs	
Key indistinguishability		✓	✓	✓	✓	✓
Explicit authentication			✓	✓	✓	✓
Probing resistance			✓	✓	✓	✓
Obfuscation of ...						
– packet lengths	✓	✓	✓	✓	✓	✓ _S
– explicit header fields		✓	✓	✓	✓	✓ _S
– entropy, directionality						✓ _S
– timing properties						
Obfuscation when pk revealed				✓	✓	✓ _{opt}
Quantum safety					✓	✓ _{opt}

Table 1: Overview of properties of obfuscated protocols. The final column indicates properties our model captures: ✓_S denotes a simulator-dependent property; ✓_{opt} an optional one.

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; in line with plans by The Tor Project to transition the many cryptographic components within Tor to quantum-safe variants. Classically, obfs4 and other FEPs employ a Diffie–Hellman-based key exchange (e.g., ntor [27]) 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 Kameleon, for mapping ML-KEM [37] 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 Kameleon 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 6).

Discussion and further challenges. Finally, we discuss (in Section 7) 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 requiring 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 [48] were the first to study public-key steganographic protocols and among them steganographic key exchange. 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 [31] studied covert authentication based on conditional KEMs and identity escrow schemes. More recently, Eldefrawy, Genise, and Jarecki [18] proposed a more efficient covert authenticated key exchange protocol based on similar building blocks and proven secure in the UC framework, in the group setting relying on a trusted manager.

Notably, parties in the covert authentication setting do not know each other’s 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 [42] 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. Meanwhile, Encrypted Client Hello [41] proposes a method for encrypting the plaintext Server Name Indication and other potentially sensitive fields in ClientHello messages. The recently proposed *stealth key exchange* [22] analyzes how to embed a covert key into the nonce of a TLS 1.3 handshake. We discuss how to apply our model to protocols that attempt to simulate a TLS 1.3 handshake (in Section 7.1).

Fully encrypted protocols. Most research on FEPs for censorship circumvention relates to classifying protocol behavior and measuring features used by censors to block traffic [5, 6, 19, 24, 49, 51]. Only recently, Fenske and Johnson [20, 21] proposed a formal model and construction for the secure channel phase of FEPs (in both the datagram 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.

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 [33] public keys as uniformly random strings. We present a systematic treatment of similar maps in the full version [28]. 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 [37] 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 $O = (\text{KGen}, \text{Encode}, \text{Decode})$ with obfuscated key length $ol \in \mathbb{N}$ consists of three algorithms:*

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

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

$$\Pr \left[\begin{array}{l} \text{Encode}(pk) = \hat{pk} \wedge \\ \text{Decode}(\hat{pk}) = pk \end{array} \mid (sk, pk, \hat{pk}) \xleftarrow{\$} \text{KGen}() \right] = 1.$$

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

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

```

KGen():
1 repeat
2    $(sk, pk) \xleftarrow{\$} \text{KGen}'()$ 
3    $\hat{pk} \leftarrow \text{Encode}(pk)$ 
4 until  $\hat{pk} \neq \perp$ 
5 return  $(sk, pk, \hat{pk})$ 

```

For keygen-then-encode obfuscated key generation schemes, we will in security reductions be interested in the probability that the first key pair (sk, pk) generated by KGen' can be successfully encoded (i.e., the probability that no resampling is necessary).

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

$$\epsilon_O^{\text{1kgensucc}} := \Pr[\text{Encode}(pk) \neq \perp \mid (sk, pk) \xleftarrow{\$} \text{KGen}'()].$$

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

Definition 2.4 (Public key uniformity). *Let $\mathcal{O}$ 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 $\mathcal{O}$ against an unbounded adversary $\mathcal{A}$ as*

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

The obfs4 protocol uses the Elligator2 encoding, which maps elliptic curve public keys to random strings. When applied to X25519 [33] keys (over Curve25519), it satisfies our definition of an obfuscated key generation scheme with first-keygen success probability $\approx \frac{1}{2}$ and public key uniformity $\approx 2^{-250.83}$. We defer the complete description and analysis of Elligator2, along with other Diffie–Hellman mappings to the full version [28].

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 uniformity for KEMs.

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

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

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

$$\Pr \left[\begin{array}{l} \text{DecodeCtxt}(\text{EncodeCtxt}(c)) = c \\ \wedge \text{DecodeCtxt}(\hat{c}) = c \end{array} \mid \begin{array}{l} (sk, pk, \hat{pk}) \xleftarrow{\$} \text{KGen}(), \\ (c, K, \hat{c}) \xleftarrow{\$} \text{Encap}(pk) \end{array} \right] = 1.$$

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

$$\Pr \left[\text{Decap}(sk, \hat{c}) \neq K \mid \begin{array}{l} (sk, pk, \hat{pk}) \xleftarrow{\$} \text{KGen}(), \\ (c, K, \hat{c}) \xleftarrow{\$} \text{Encap}(pk) \end{array} \right] \leq \delta_{\text{OKEM}}.$$

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 $\text{pkcoll}_{\text{OKEM}}(n) := \Pr[pk_i = pk_j \wedge i \neq j \mid (sk_i, pk_i, \hat{pk}_i) \xleftarrow{\$} \text{OKEM}.\text{KGen}() \text{ for } i \in [1, n]]$.

As with obfuscated key generation schemes, an obfuscated KEM may follow an analogous “encapsulate-then-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 ((Keygen/Encapsulate-then-encode obfuscated KEM). *Let $\text{KEM}' = (\text{KGen}, \text{Encap}', \text{Decap}')$ 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-then-encode obfuscated KEM $\text{OKEM} = (\text{KGen}, \text{Encode}, \text{Decode}, \text{Encap}, \text{Decap}, \text{EncodeCtxt}, \text{DecodeCtxt})$ with:*

$\text{Encap}(pk)$:	$\text{Decap}(sk, \hat{c})$:
1 repeat	1 $c \leftarrow \text{DecodeCtxt}(\hat{c})$
2 $(c, K) \xleftarrow{\$} \text{Encap}'(pk)$	2 return $\text{Decap}'(sk, c)$
3 $\hat{c} \xleftarrow{\$} \text{EncodeCtxt}(c)$	
4 until $\hat{c} \neq \perp$	
5 return $(c, K, \hat{c})$	

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

Similar to keygen-then-encode obfuscated key generation schemes, for encapsulate-then-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 an encapsulate-then-encode obfuscated KEM based on $\text{KEM}' = (\text{KGen}', \text{Encap}', \text{Decap}')$. The first-encap success probability of OKEM is*

$$\epsilon_{\text{OKEM}}^{\text{encsucc}} := \Pr \left[\text{EncodeCtxt}(c) \neq \perp \mid \begin{array}{l} (sk, pk) \xleftarrow{\$} \text{KGen}'(), \\ (c, K) \xleftarrow{\$} \text{Encap}'(pk) \end{array} \right].$$

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

$$\text{Adv}_{\text{OKEM}}^{\text{ctxt-unif}}(\mathcal{A}) := 2 \cdot \Pr \left[\mathcal{A}(\hat{pk}, \hat{c}_b) = b \mid \begin{array}{l} b \xleftarrow{\$} \{0, 1\}, \hat{c}_0 \xleftarrow{\$} \{0, 1\}^{cl}, \\ (sk, pk, \hat{pk}) \xleftarrow{\$} \text{OKEM}.\text{KGen}(), \\ (c_1, K_1, \hat{c}_1) \xleftarrow{\$} \text{OKEM}.\text{Encap}(pk) \end{array} \right] - 1.$$

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 extend 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 [35, 53]. We show that for keygen/encapsulate-then-encode obfuscated KEMs (Definition 2.8), the two notions

$\mathbf{G}_K^{\text{IND-CCA}}(\mathcal{A}):$ <pre> 1 $b \xleftarrow{\\$} \{0, 1\}$ 2 $(sk, pk, \overset{\boxed{\text{---}}}{pk}) \xleftarrow{\\$} \text{KGen}()$ 3 $(c^*, K_1^*, \overset{\boxed{\text{---}}}{c_1^*}) \xleftarrow{\\$} \text{Encap}(pk)$ 4 $K_0^* \xleftarrow{\\$} \mathcal{K}$ 5 $x \leftarrow c^* \overset{\boxed{\text{---}}}{\text{---}}$ 6 $b' \xleftarrow{\\$} \mathcal{A}^{\text{Decap}_x(\cdot)}(pk, c^*, K_b^*, \overset{\boxed{\text{---}}}{c_b^*})$ 7 return $\llbracket b = b' \rrbracket$ </pre> $\text{Decap}_x(c):$ <pre> 1 if $c = x$ then return $\perp$ 2 $K \leftarrow \text{Decap}(sk, c)$ 3 return K </pre>	$\mathbf{G}_{K(S)}^{\text{SPR-CCA}}(\mathcal{A}):$ <pre> 1 $b \xleftarrow{\\$} \{0, 1\}$ 2 $(sk, pk, \overset{\boxed{\text{---}}}{pk}) \xleftarrow{\\$} \text{KGen}()$ 3 $(c_1^*, K_1^*, \overset{\boxed{\text{---}}}{c_1^*}) \xleftarrow{\\$} \text{Encap}(pk)$ 4 $c_0^* \xleftarrow{\\$} \mathcal{S}$ 5 $\overset{\boxed{\text{---}}}{c_0^*} \xleftarrow{\\$} \{0, 1\}^{\text{cl}}; c_0^* \leftarrow \text{DecodeCtxt}(\overset{\boxed{\text{---}}}{c_0^*})$ 6 $K_0^* \xleftarrow{\\$} \mathcal{K}$ 7 $x \leftarrow c_b^* \overset{\boxed{\text{---}}}{\text{---}}; x \leftarrow c_b^* \overset{\boxed{\text{---}}}{\text{---}}$ 8 $b' \leftarrow \mathcal{A}^{\text{Decap}_x(\cdot)}(pk, c_b^*, K_b^*, \overset{\boxed{\text{---}}}{c_b^*})$ 9 return $\llbracket b = b' \rrbracket$ </pre>
--	---

Figure 1: Security games for IND-CCA and SPR-CCA security of a regular KEM or obfuscated KEM $K = (\text{KGen}, \text{Encap}, \text{Decap}, \dots)$ with key space $\mathcal{K}$. Code in dashed boxes is only included for an obfuscated KEM; the simulator $\mathcal{S}$ is only involved for regular KEMs.

are implied by the underlying KEM achieving them and the obfuscated KEM's first-keygen/encap success probability and ciphertext uniformity. This in particular will apply to our obfuscated KEM ML-Kameleon which we introduce in Section 2.4.

Definition 2.11 ((Obfuscated) KEM security). Let $K = (\text{KGen}, \text{Encap}, \text{Decap}, \dots)$ be a KEM or obfuscated KEM with key space $\mathcal{K}$ and $\mathbf{G}_{K(S)}^{\text{goal-CCA}}$ for $\text{goal} \in \{\text{IND}, \text{SPR}\}$ be defined as in Figure 1. We say that K is (t, ϵ, q) -goal-CCA-secure, with respect to a simulator $\mathcal{S}$ when $\text{goal} = \text{SPR}$, 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

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

We also consider two weaker notions: IND-1CCA security, restricting $\mathcal{A}$ in the IND-CCA game to a single Decap call, and SPR-CPA security, omitting the Decap oracle from the SPR-CCA game.

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

Theorem 2.12 (Keygen/encapsulate-then-encode obfuscated KEM IND-CCA security). Let OKEM be a keygen/encapsulate-then-encode obfuscated KEM based on a regular KEM K 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

$$\text{Adv}_{\text{OKEM}}^{\text{IND-CCA}}(\mathcal{A}) \leq 1/\epsilon_{\text{OKEM}}^{\text{1kgsucc}} \cdot 1/\epsilon_{\text{OKEM}}^{\text{1encsucc}} \cdot \text{Adv}_K^{\text{IND-CCA}}(\mathcal{B}).$$

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

Theorem 2.13 (Keygen/encapsulate-then-encode obfuscated KEM SPR-CCA security). Let OKEM be a keygen/encapsulate-then-encode obfuscated KEM based on a regular KEM K 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

$$\begin{aligned} \text{Adv}_{\text{OKEM}}^{\text{SPR-CCA}}(\mathcal{A}) &\leq 1/\epsilon_{\text{OKEM}}^{\text{1kgsucc}} \cdot 1/\epsilon_{\text{OKEM}}^{\text{1encsucc}} \cdot \text{Adv}_{K, \mathcal{S}_{\text{unif}}}^{\text{SPR-CCA}}(\mathcal{B}_1) \\ &\quad + \text{Adv}_{\text{OKEM}}^{\text{ctxt-unif}}(\mathcal{B}_2), \end{aligned}$$

where $\mathcal{S}_{\text{unif}}$ is a “uniform-encapsulation” simulator for KEM.

We defer the proofs of both theorems to the full version [28]; the main idea in both proofs is to quantify how much rejection sampling narrows the space of public keys/ciphertexts.

2.4 Obfuscating ML-KEM with Kameleon

Prior work constructed encodings for Diffie–Hellman public keys to random bytestrings [13, 26, 52], 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 [37] public keys and ciphertexts, and can be combined with ML-KEM to create an obfuscated KEM. We call our encoding scheme Kameleon and give its algorithms in Figure 2.¹

Notation. For a vector $\mathbf{a}$ of length $\text{len}(\mathbf{a})$, $\mathbf{a}[i]$ denotes the i th 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 i th 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 i th and j th 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 and hence can simply be concatenated in the encoded public key, 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, where every pair of coefficients is mapped to three 8-bit integers. Every 12th bit in the output is then biased toward 0.

We use a combination of accumulation and rejection sampling, in line with the keygen-then-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. 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-keygen success probability is the probability that the highest-order bit after accumulation is 0:

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

¹Our encoding also works for Kyber [8] which has identical public keys and ciphertexts. Naturally, we call that combination Kyber-Kameleon [16].

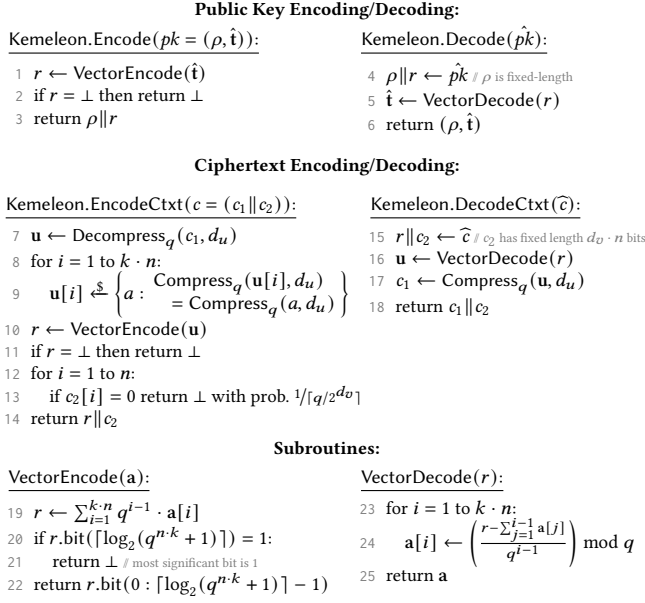


Figure 2: Kemeleon public key and ciphertext encoding and decoding algorithms.

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, where $q = 3329$, $n = 256$, $k = 2$, 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 . For uniform $\hat{\mathbf{t}} \in (\mathbb{Z}_q^n)^k$, the statistical distance of the encoded public key from uniform ol -bit strings is 0, since the Kemeleon output covers the entire space uniformly (recall that ρ is already uniformly random). The advantage of any adversary against public key uniformity now depends on the hardness of the Module-LWE problem, as distinguishing the encoded bitstring from a random string can be accomplished by observing that the vector $\hat{\mathbf{t}}$ in the public key is of the form $\hat{\mathbf{A}} \cdot \hat{\mathbf{s}} + \hat{\mathbf{e}}$, where $\hat{\mathbf{A}}$ is generated from the seed ρ .

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, respectively). The Compress and Decompress algorithms are defined in [37, Section 4.2.1] as follows, and act component-wise: $\text{Compress}_q(x, d) := \lceil (2^d/q) \cdot x \rceil \bmod 2^d$ (taking the canonical representative in $[0, 2^d)$) and $\text{Decompress}_q(x, d) := \lceil (q/2^d) \cdot x \rceil$.

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 $c_2 = \text{Compress}_q(v, d_v)$, has a similar 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 a slightly higher probability than other outputs (increased by $1/[q/2^{d_v}]$). In this case, we can perform rejection sampling: for each coefficient of c_2 equal to 0, reject (output $\perp$) with probability $1/[q/2^{d_v}]$. 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 $(1 - 1/(2^{d_v} \cdot [q/2^{d_v}]))^n$, 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 $\text{VectorEncode}(u) \neq \perp$ in line 5 of EncodeCtxt and that the second element is not rejected:

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

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 keygen/encapsulate-then-encode obfuscated KEM (Definition 2.8) based on ML-KEM and using the Kemeleon public key and ciphertext encoding/decoding algorithms from Figure 2.

We establish the following bounds for the public key and ciphertext uniformity of ML-Kemeleon, the proof of which we give in the full version [28].

Lemma 2.14 (pk-unif of ML-Kemeleon). *For any adversary $\mathcal{A}$ against the public key uniformity (Definition 2.4) of ML-Kemeleon, we give an algorithm $\mathcal{B}$, such that*

$$\text{Adv}_{\text{ML-Kemeleon}}^{\text{pk-unif}}(\mathcal{A}) \leq 1/\epsilon_{\text{Kemeleon}}^{\text{encsucc}} \cdot \text{Adv}_{\text{ML-KEM}}^{\text{M-LWE}}(\mathcal{B}).$$

Lemma 2.15 (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 S_{compr} that samples uniformly random ciphertexts $(\mathbf{u}, v)$ before compression and outputs*

$c = \text{Compress}_q(u, d_u) \parallel \text{Compress}_q(v, d_v)$, such that

$$\text{Adv}_{\text{ML-Kameleon}}^{\text{ctxt-unif}}(\mathcal{A}) \leq 1/\epsilon_{\text{Kameleon}}^{\text{1kgensucc}} \cdot 1/\epsilon_{\text{Kameleon}}^{\text{1ensucc}} \cdot \text{Adv}_{\text{ML-KEM}, S_{\text{compr}}}^{\text{SPR-CPA}}(\mathcal{B}).$$

3 The obfs4 Protocol

The obfs4 protocol, specified in [45]², is separated into two distinct phases: a *key exchange* phase and a *data transfer* phase. We are interested specifically in the key exchange protocol, which is shown in Figure 3. The intent of obfs4's key exchange is to establish a shared key between client and server which is forward secret and explicitly authenticated by the server. This is generally accomplished via an ntor handshake component [27], which is the traditional implicitly authenticated Diffie–Hellman key exchange protocol used in Tor. The additional steps of the protocol are to ensure obfuscation properties, i.e., that the messages exchanged are of random length and consist of random byte strings. 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 keygen-then-encode obfuscated key generation, in which it samples an ephemeral X25519 [33] 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.

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 [27] 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 . 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 . Here, $auth$ serves to provide authentication of the server, and M_S is necessary for the client to parse the message, because the server may continue sending data following the final authentication tag.

Upon receipt, the client similarly verifies both MAC values and then performs the remainder of the client-side operations of the

Server key generation/setup

$NodeID \xleftarrow{\$} \{0, 1\}^{160}; (b, B) \xleftarrow{\$} \text{KGen}_{\text{X25519}}(); st.S_{MAC} \leftarrow \emptyset$
 return $((b, NodeID), (B, NodeID), st)$

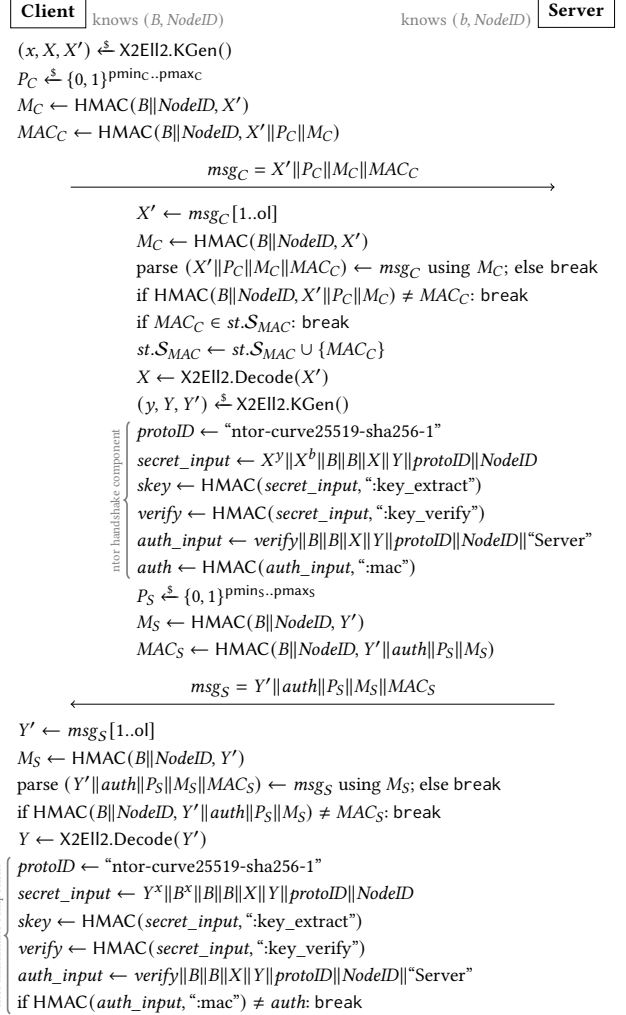


Figure 3: The obfs4 [45] obfuscated key exchange protocol.

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 S_{MAC} per epoch, and clear that storage for an epoch once the one after next 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

²We base our analysis on the lyrebird implementation from The Tor Project [39].

are used in practice by censors to classify obfuscated traffic (discussed in Section 7.2). Our goals are also motivated by the explicit threat models of FEPs, such as those in [45, §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 [45, §2]:

“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 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 7.2. 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 [47]: *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 [19] and evidence of the efficacy of

probing against FEPs [24], the following requirement was included in the obfs4 threat model [45, §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 [45, §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 $role \in \{\text{initiator, responder}\}$.
- **Run**($\pi_u^i, st_u, sk_u, pk_v, m$) $\xrightarrow{\$}$ (π_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 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 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 i th session owned by user u . Each user u is assigned a role, $u.role \in \{\text{initiator, responder}\}$, 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 following session variables in sans-serif font are accessible by the security game only:

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

³The steganographic key exchange security notion from [48] can be seen as a single CHALLEXEC oracle run (one real-or-random trace, with real-or-random key).

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* [17] 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 $G_{KE,S}^{ObfKE}(\mathcal{A})$ be the obfuscated key exchange security game wrt. a simulator S defined in Figure 4 for an adversary $\mathcal{A}$. We define the advantage of $\mathcal{A}$ in breaking the ObfKE security of KE as $\text{Adv}_{KE,S}^{ObfKE}(\mathcal{A}) := 2 \cdot \Pr[G_{KE,S}^{ObfKE}(\mathcal{A}) \Rightarrow 1] - 1$. We distinguish two flavors of ObfKE, capturing regular (rObfKE) and strong (sObfKE) obfuscation through different ObfFresh predicates (Figure 4); 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) CHALLENGE queries against some pre-declared server.

We denote these variants as ObfKE-1 (for a single CHALLENGE query) and ObfKE-1* (for multiple CHALLENGE queries). These are weaker versions of the main game, asking only for selective security in a single-challenge setting. Yet we show 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 $(n_s + n_r \cdot q_C)$ 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 CHALLENGE queries the adversary makes. For ObfKE-1*, allowing multiple CHALLENGE queries to the pre-declared server, the loss is only $(n_s + n_r)$.

For space reasons, we defer the single-challenge definitions and hybrid proofs for ObfKE-1 and ObfKE-1* to the full version [28].

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 [38] being hard for X25519, modeling HMAC as a random oracle, and the public key uniformity of keygen-then-encode obfuscated X25519 [33] 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 S_{obfs4} which outputs two uniformly random messages of length corresponding to the obfs4 message elements including random-length padding:

S_{obfs4} :

```

1  $r_C \xleftarrow{\$} [\text{pmin}_C, \text{pmax}_C] ; r_S \xleftarrow{\$} [\text{pmin}_S, \text{pmax}_S] \quad \# \text{ random-length client/server padding}$ 
2  $m_1 \xleftarrow{\$} \{0, 1\}^{\text{ol}+512+r_C} \quad \# 1x \text{ encoded DH} + 2x \text{ HMAC outputs} + \text{client padding}$ 
3  $m_2 \xleftarrow{\$} \{0, 1\}^{\text{ol}+768+r_S} \quad \# 1x \text{ encoded DH} + 3x \text{ HMAC outputs} + \text{server padding}$ 
4 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 given in the full version [28].

Theorem 5.1. Let obfs4 be defined as in Figure 3. 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

$$\begin{aligned} \text{Adv}_{obfs4, S_{obfs4}}^{rObfKE}(\mathcal{A}) \leq & 2 \cdot \left(\frac{n_s^2 + n_r^2}{q} + \frac{n_r^2 + 3q_{RO} \cdot n_r}{2^{160}} + \frac{q_S}{2^{256}} \right. \\ & + n_s n_r \cdot 1/\epsilon_{X2Ell2}^{1\text{kgensucc}} \cdot \left(\text{Adv}_{X25519}^{\text{GapDH}}(\mathcal{B}_1) + \frac{1}{2^{256}} \right) \\ & + (n_s + n_r) \cdot \left(n_s \cdot (1/\epsilon_{X2Ell2}^{1\text{kgensucc}})^2 \cdot \text{Adv}_{X25519}^{\text{GapDH}}(\mathcal{B}_2) \right. \\ & \left. \left. + \frac{3q_{RO}}{2^{160}} + 2q_C \cdot \text{Adv}_{X2Ell2}^{\text{pk-unif}}(\mathcal{B}_3) \right) \right), \end{aligned}$$

where $\mathcal{A}$ makes at most q_S, q_C , and q_{RO} queries to its SEND oracle, CHALLENGE 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 keygen-then-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 4 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). We exclude $NodeID$ collisions ($n_r^2/2^{160}$) and then rule out $\mathcal{A}$ querying the HMAC random oracle on an uncompromised $NodeID$ value ($3q_{RO} \cdot n_r/2^{160}$). In particular, the challenger samples 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 $u.role = \text{responder}$. However, from the

$\text{GObfKE}_{\text{KE},S}(\mathcal{A})$ INITIALIZE: 1 time $\leftarrow 0$; users $\leftarrow \emptyset$ 2 $b \xleftarrow{\$} \{0,1\}$ 3 Probed $\leftarrow \text{false}$ NEWUSER ($id, role$): 4 $u \leftarrow ++\text{users}$ 5 $u.role \leftarrow role$ 6 $(pk_u, sk_u, st_u) \xleftarrow{\$} \text{Setup}(id, role)$ 7 $revsk_u \leftarrow \infty$; $revpk_u \leftarrow \infty$ SEND (u, i, m): 30 $f \leftarrow \llbracket \pi_u^i = \perp \rrbracket$ // First SEND to this session? 31 if $\pi_u^i = \perp$ and $u.role = \text{initiator}$ then 32 $\pi_u^i.peerid \leftarrow m$; $m \leftarrow \varepsilon$ // "Virtual" first message to initiator sets peer id 33 if $\pi_u^i.status \in \{\text{accepted}, \text{rejected}\}$ then return $\perp$ 34 $(\pi_u^i, st_u, m') \xleftarrow{\$} \text{Run}(\pi_u^i, st_u, sk_u, pk_{\pi_u^i.peerid}, m)$ 35 if $\pi_u^i.status = \text{accepted}$ then $\pi_u^i.t_{acc} \leftarrow ++\text{time}$ 36 if f then // Upon first message sent/received... 37 if $u.role = \text{initiator}$ then 38 $\mathcal{F}_{\pi_u^i.peerid} \leftarrow \mathcal{F}_{\pi_u^i.peerid} \cup \{m'\}$ // Record initiators' first messages (per responder) 39 if $u.role = \text{responder}$ and $m' \neq \varepsilon$ then 40 if $m \notin \mathcal{F}_u$ and $\neg revpk_u$ then Probed $\leftarrow \text{true}$ // Response to a non-initiator-first message by a non- pk -revealed responder 41 $\mathcal{F}_u \leftarrow \mathcal{F}_u \setminus \{m\}$ // Consider m "consumed" 42 return $(\pi_u^i.status, m')$	REVSESSIONKEY (u, i): 8 if $\pi_u^i = \perp$ or $\neg \pi_u^i.t_{acc}$ then return $\perp$ 9 $\pi_u^i.revealed \leftarrow \text{true}$ 10 return $\pi_u^i.skey$ REVSECRETKEY (u): 11 if $revsk_u$ then return $\perp$ 12 $revsk_u \leftarrow ++\text{time}$ 13 if $\neg revpk_u$ then $revpk_u \leftarrow \text{time}$ // Consider pk_u revealed, too 14 return sk_u REVPUBLICKEY (u): 15 if $revpk_u$ then return $\perp$ 16 $revpk_u \leftarrow ++\text{time}$ 17 return pk_u	TEST (u, i): 18 if $\pi_u^i = \perp$ or $\neg \pi_u^i.t_{acc}$ or $\pi_u^i.tested$ then 19 return $\perp$ 20 $\pi_u^i.tested \leftarrow \text{true}$ 21 $k_1 \leftarrow \pi_u^i.skey$ 22 $k_0 \xleftarrow{\$} \text{KE.KS}$ 23 return k_b FINALIZE (b'): 24 if $\neg \text{Sound}$ then $b' \leftarrow b$ 25 if $\neg \text{ExplicitAuth}$ then $b' \leftarrow b$ 26 if Probed then $b' \leftarrow b$ 27 if $\neg \text{Fresh}$ then $b' \leftarrow 0$ 28 if $\neg \text{ObfFresh}$ then $b' \leftarrow 0$ 29 return $\llbracket b = b' \rrbracket$ CHALLENGE (u, v): 43 $\pi_u \leftarrow \pi_v \leftarrow \perp$ // Temporary initiator and responder sessions 44 if $u.role \neq \text{initiator}$ or $v.role \neq \text{responder}$ then return $\perp$ 45 $trans_1 \leftarrow ()$; $\pi_u.peerid \leftarrow v$; $p \leftarrow u$; $m \leftarrow \varepsilon$ 46 repeat // Execute entire protocol and collect real transcript 47 $(\pi_p, st_p, m) \xleftarrow{\$} \text{Run}(\pi_p, st_p, sk_p, pk_{\pi_p.peerid}, m)$ 48 $trans_1 \leftarrow trans_1 \parallel (m)$ 49 if $p = u$ then $p \leftarrow v$ else $p \leftarrow u$ // Switch parties 50 until $\pi_u.status = \pi_v.status = \text{accepted}$ 51 $trans_0 \xleftarrow{\$} \mathcal{S}()$ // Simulated transcript 52 $k_1 \leftarrow \pi_u.skey$; $k_0 \xleftarrow{\$} \text{KE.KS}$ // Real-or-random session key 53 $chall_v \leftarrow \text{true}$ // Mark server v as challenged 54 return $(trans_0, k_b)$
Fresh: 1 for each $\pi_u^i : \pi_u^i.tested$ 2 if $\pi_u^i.revealed$ then return false // tested session may not be revealed 3 if $\exists \pi_v^j : \pi_v^j.sid = \pi_u^i.sid \wedge (\pi_v^j.tested \vee \pi_v^j.revealed)$ then 4 return false // tested session's partnered session may not be tested or revealed 5 if $u.role = \text{initiator} \wedge revsk_{\pi_u^i.peerid} \leq \pi_u^i.t_{acc} \wedge \neg \exists \pi_v^j : \pi_u^i : (\pi_u^i.sid = \pi_v^j.sid)$ 6 return false // initiators: forward secrecy (peer's sk unrevealed prior to acceptance) or passive execution 7 if $u.role = \text{responder} \wedge \neg \exists \pi_v^j : \pi_u^i : (\pi_u^i.cid = \pi_v^j.cid \wedge v.role = \text{initiator})$ 8 return false // responders: security only for passive executions (as initiators are unauthenticated) 9 return true	ExplicitAuth: // Explicit authentication of responders to initiators 1 $\forall \pi_u^i : (u.role = \text{initiator} \wedge \pi_u^i.t_{acc} < revsk_{\pi_u^i.peerid})$ $\implies \exists \pi_v^j : v = \pi_u^i.peerid \wedge \pi_u^i.sid = \pi_v^j.sid$	Sound: 1 if $\exists$ distinct $\pi_u^i, \pi_v^j, \pi_w^k : \pi_u^i.sid = \pi_v^j.sid = \pi_w^k.sid \neq \perp$ then 2 return false // no triple sid match 3 if $\exists \pi_u^i, \pi_v^j : \pi_u^i.sid = \pi_v^j.sid \neq \perp \wedge u.role = v.role$ then 4 return false // partnering implies different roles 5 if $\exists \pi_u^i, \pi_v^j : \pi_u^i.sid = \pi_v^j.sid \neq \perp \wedge \pi_u^i.cid \neq \pi_v^j.cid$ then 6 return false // partnering implies same contributive identifiers 7 if $\exists \pi_u^i, \pi_v^j : \pi_u^i.sid = \pi_v^j.sid \neq \perp \wedge u.role = \text{initiator} \wedge \pi_u^i.peerid \neq v$ then 8 return false // partnering implies agreement on responder ID 9 if $\exists \pi_u^i, \pi_v^j : \pi_u^i.sid = \pi_v^j.sid \neq \perp \wedge \pi_u^i.skey \neq \pi_v^j.skey$ then 10 return false // partnering implies same key 11 return true
ObfFresh: 1 if $\exists v : revpk_v \wedge chall_v$ then // Regular obfuscation (rObfKE) 2 if $\exists v : revsk_v \wedge chall_v$ then // Strong obfuscation (sObfKE) 3 return false // Challenge pk revealed (fwd-secret: prior to challenge) 4 return true		

Figure 4: Obfuscated key exchange security (ObfKE) game (top) capturing key indistinguishability, explicit server authentication, (regular (rObfKE) or strong (sObfKE)) obfuscation with respect to a simulator S , and probing resistance, via predicates (bottom) Fresh, ExplicitAuth, ObfFresh, and flag Probed, respectively.

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.

Now Probed is set if a non-initiator-first message m yields a reply by an unrevealed responder. Such a message m is either rejected due to replay checks against MAC values, or is different from all honest initiators' first messages sent to this responder. In the latter case, m contains the client MAC MAC_C , which is now a random 256-bit value, leaving $\mathcal{A}$ a $q_S/2^{256}$ guessing chance.

A second branch prevents explicit authentication (ExplicitAuth) from being violated. We first guess the first session violating ExplicitAuth, π^* , and its peer, v^* , (losing a factor $n_S \cdot n_r$), and rule out the case that its DH share is not encodable (losing a factor $1/\epsilon_{\text{X2ElI2}}^{\text{kgensucc}}$). Then, we can embed a GapDH challenge in that session's DH public key X and the partner's DH public key B (term $\text{Adv}_{\text{X2519}}^{\text{GapDH}}(\mathcal{B}_1)$). Specifically, we replace the outputs of the random oracle HMAC using as input *secret_input* containing DH(X, B) with uniformly random values. This means the *verify* value derived in the target session 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 π^* and its

peer's long-term public key B . Then $\mathcal{B}_1$ simulates 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 $\text{DDH}(Z, B, C)$. Unless $\mathcal{A}$ makes a query involving the DH secret $\text{DH}(X, B)$ prior to π^* accepting, it cannot detect the replacements upon input secret_input (including verify^*). If $\mathcal{A}$ makes such a query, $\mathcal{B}_1$ is able to detect this (using its DDH oracle) and wins the GapDH game by outputting the CDH solution $\text{DH}(X, B)$.

Finally, the target session π^* computes the authentication value, auth , using HMAC on input a uniformly random value $\text{verify}^* \in \{0, 1\}^{256}$ unknown to $\mathcal{A}$ prior to π^* accepting. 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 π^* (n_s loss) and abort if the DH share in either session is not encodable ($(1/\epsilon_{\text{X2Ell2}}^{\text{kgensucc}})^2$ loss). We then replace the outputs of the random oracle HMAC on input secret_input containing $\text{DH}(X, Y)$ 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 . 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 $\text{DH}(X, Y)$, it cannot detect the replacement, and if it does, $\mathcal{B}_2$ detects this and outputs the CDH solution. This incurs the term $\text{Adv}_{\text{X2519}}^{\text{GapDH}}(\mathcal{B}_2)$ and results in the test session's key being random, which concludes this case.

For Case II ($\mathcal{A}$ wins via CHALLENGE 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 $3q_{\text{RO}}/2^{160}$). In particular, the challenger samples 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 CHALLENGE sessions with server p . 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 $\text{HMAC}(B||\text{NodeID}, \cdot)$, in secret_input , and in auth_input , each random oracle query might match with one of the three input types. Next, instead of computing the ephemeral DH shares of sessions in the CHALLENGE oracle as $(x, X, X') \xleftarrow{\$} \text{X2Ell2.KGen}()$ and $(y, Y, Y') \xleftarrow{\$} \text{X2Ell2.KGen}()$, we sample $X' \xleftarrow{\$} \{0, 1\}^{\text{ol}}$ and $Y' \xleftarrow{\$} \{0, 1\}^{\text{ol}}$, and then compute the DH shares as $X \leftarrow \text{X2Ell2.Decode}(X')$ and $Y \leftarrow \text{X2Ell2.Decode}(Y')$. 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 $\text{Decode}(\hat{pk})$

in place of X resp. Y . This can be bounded by the uniformity of X2519 public keys encoded with Elligator2 via a hybrid argument of all q_C many CHALLENGE queries, i.e., $2q_C \cdot \text{Adv}_{\text{X2Ell2}}^{\text{pk-unif}}(\mathcal{B}_3)$. At this point, the protocol messages in CHALLENGE sessions are uniformly distributed like outputs of the simulator $\mathcal{S}_{\text{obfs4}}$, concluding this case and the proof. $\square$

Achieving strong obfuscation. 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 In the full version [28], 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.

The core idea is to use the DH ephemeral/long-term secret for computing MAC tags, which comes with some decoding overhead but sObfKE security for st-obfs under the same assumptions as for obfs4; see the full version [28] for the details. There, we also discuss a variant of strong obfuscation akin to forward secrecy.

6 Quantum-safe Obfuscated Key Exchange

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

6.1 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. At a high-level, we have replaced the ephemeral-ephemeral and ephemeral-static DH exchanges in obfs4 with ephemeral and static KEM encapsulations, and carefully adapted the key schedule inspired by HKDF's Extract-then-Expand paradigm [32] to obtain security under standard-model assumptions. The use of KEMs in place of the elliptic curve key exchange operations enables post-quantum security of the protocol. Aside from the use of KEMs, pq-obfs is largely modelled after obfs4 and ntor, and includes similar MAC tags and random-length padding for the same purposes as in obfs4. Strong obfuscation is achieved via an ephemeral key input to the computations of MAC tags, similar to the approach in st-obfs. 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 5. 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 $F: X \times \mathcal{Y} \rightarrow \mathcal{Z}$ 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 $F'(y, x) = F(x, y)$. We say that F is a *swap-PRF* if

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

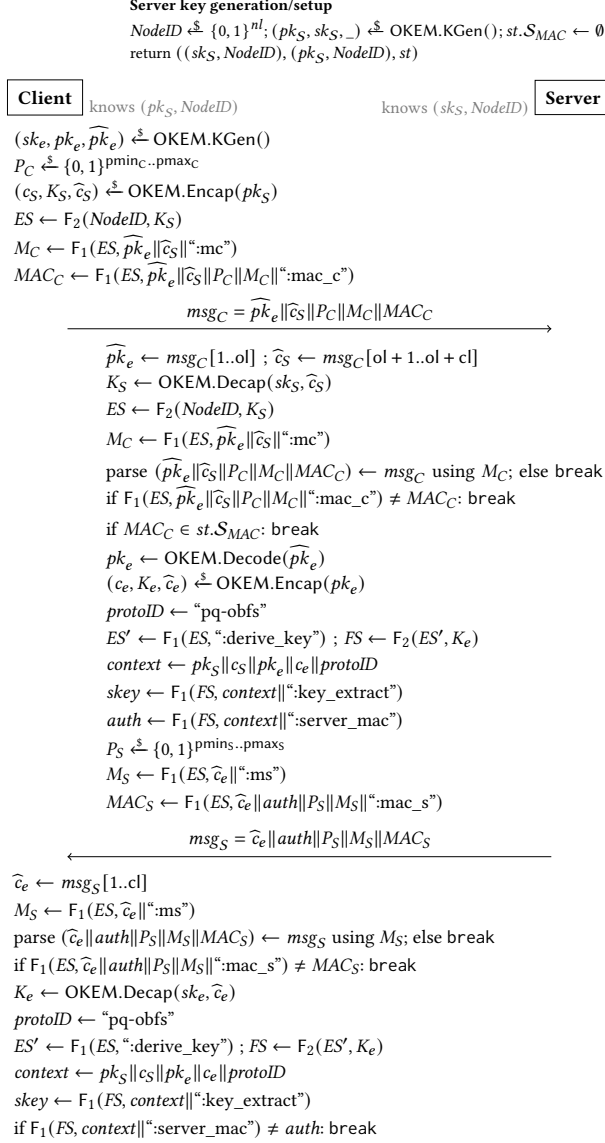


Figure 5: Our post-quantum obfuscated key exchange protocol pq-obfs with strong obfuscation.

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*.

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-Kameleon (ML-KEM [37] with our Kameleon encoding of public keys and ciphertexts as in Section 2.4). The IND-CCA and SPR-CCA security of ML-KEM, established in [15, 30, 35], translate to ML-Kameleon, with a small loss from its first-keygen/encap 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 [10].

Performance. Our pq-obfs protocol is structurally close to obfs4: client and server send one message each (one round-trip) to establish ephemeral-ephemeral and ephemeral-static secrets, and from those derive session keys and authentication tags. In terms of performance, the difference is dominated by the costs arising from replacing elliptic-curve DH with a KEM, since the other changes are small and efficient (e.g., instantiating key derivation with HMAC). For X25519 vs. Kyber/ML-KEM, this primarily means that pq-obfs will have a larger communication overhead for sending public keys and ciphertexts (781 and 877 bytes for Kameleon-encoded ML-KEM-512 vs. twice 32 bytes for X25519), while computations can actually be faster in practice [50].

6.2 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 $S_{pq-obfs}$ which outputs two uniformly random messages of length corresponding to the pq-obfs message elements including random-length padding:

$S_{pq-obfs}$:

- 1 $rc \xleftarrow{\$} [pminc, pmaxc]; rs \xleftarrow{\$} [pmins, pmaxs] \text{ // random-length client/server padding}$
- 2 $m_1 \xleftarrow{\$} \{0, 1\}^{ol+cl+2 \cdot fl_1+rc} \text{ // 1x encoded pk + 1x encoded ctxt + 2x } F_1 \text{ outputs + client padding}$
- 3 $m_2 \xleftarrow{\$} \{0, 1\}^{cl+3 \cdot fl_1+rs} \text{ // 1x encoded ctxt + 3x } F_1 \text{ outputs + server padding}$
- 4 return (m_1, m_2)

Session and contributive identifiers. We set the session identifier 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.

We now give the theorem statement and a proof sketch; the complete proof is given in the full version [28]. Our design allows the proof to proceed via a sequence of reductions to (dual-)PRF security of F_1 and F_2 , applications of IND-CCA and SPR-CCA security, and uniformity properties of the obfuscated KEM; we do not need to rely on a (quantum) random oracle.

Theorem 6.1. Let pq-obfs be defined as in Figure 5. For any sObfKE adversary $\mathcal{A}$ against pq-obfs, we give algorithms $\mathcal{B}_1 \dots \mathcal{B}_{17}$ in the proof such that

$$\begin{aligned} \text{Adv}_{pq-obfs, S_{pq-obfs}}^{\text{sObfKE}}(\mathcal{A}) &\leq 2 \cdot \left(\text{pkcoll}_{\text{OKEM}}(n_s + n_r) + 2n_s \cdot \delta_{\text{OKEM}} \right. \\ &\quad + n_s n_r \cdot \left(\text{Adv}_{F_2}^{\text{PRF}}(\mathcal{B}_1) + \text{Adv}_{F_1}^{\text{PRF}}(\mathcal{B}_2) + \frac{2}{2^{fl_1}} + \text{Adv}_{\text{OKEM}}^{\text{IND-CCA}}(\mathcal{B}_3) \right. \\ &\quad \left. \left. + \text{Adv}_{F_2}^{\text{swap-PRF}}(\mathcal{B}_4) + \text{Adv}_{F_2}^{\text{PRF}}(\mathcal{B}_5) + \text{Adv}_{F_1}^{\text{PRF}}(\mathcal{B}_6) + \text{Adv}_{F_1}^{\text{PRF}}(\mathcal{B}_7) \right) \right) \\ &\quad + (n_s + n_r q_C) \cdot \left(n_s \left(\text{Adv}_{\text{OKEM}}^{\text{IND-1CCA}}(\mathcal{B}_8) + \text{Adv}_{F_2}^{\text{swap-PRF}}(\mathcal{B}_9) + \text{Adv}_{F_1}^{\text{PRF}}(\mathcal{B}_{10}) \right) \right) \end{aligned}$$

$$+ \text{Adv}_{\text{OKEM}}^{\text{SPR-CCA}}(\mathcal{B}_{11}) + \text{Adv}_{F_2}^{\text{swap-PRF}}(\mathcal{B}_{12}) + \text{Adv}_{F_1}^{\text{PRF}}(\mathcal{B}_{13}) + \text{Adv}_{F_2}^{\text{PRF}}(\mathcal{B}_{14}) \\ + \text{Adv}_{F_1}^{\text{PRF}}(\mathcal{B}_{15}) + \text{Adv}_{\text{OKEM}}^{\text{ctxt-unif}}(\mathcal{B}_{16}) + \text{Adv}_{\text{OKEM}}^{\text{pk-unif}}(\mathcal{B}_{17})) \Bigg),$$

where $\mathcal{A}$ makes at most q_C CHALLENGE 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 $\text{pkcoll}_{\text{OKEM}}(n_s + n_r)$) and assume correct decapsulation in all sessions ($2n_s \cdot \delta_{\text{OKEM}}$), 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 this, we observe that Probed is set if a non-initiator-first message m yields a reply by an unrevealed responder. Such a message m is either rejected due to the replay check, or it is different from all honest initiators' first messages sent to this responder. In the second case, m contains the target client MAC value MAC_C^* which is a random fl_1 -bit value unknown to $\mathcal{A}$. The adversary $\mathcal{A}$ can therefore only guess MAC_C^* ; the corresponding bound is $\text{Adv}_{F_2}^{\text{PRF}}(\mathcal{B}_1) + \text{Adv}_{F_1}^{\text{PRF}}(\mathcal{B}_2) + 1/2^{fl_1}$.

A second branch prevents explicit authentication (ExplicitAuth) being violated. We guess the first session violating ExplicitAuth, π^* , and its peer, v^* , (losing a factor $n_s \cdot n_r$). Then 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 as follows (term $\text{Adv}_{\text{OKEM}}^{\text{IND-CCA}}(\mathcal{B}_3)$). Let $(c_S, K_S, \widehat{c}_S) \xleftarrow{\$} \text{OKEM.Encap}(pk_S)$ be the encapsulation computed in the target session π^* with the long-term public key of v^* . We replace the long-term KEM key K_S with a uniformly random $\tilde{K}_S$ in π^* . All values derived from K_S in π^* use the randomized value $\tilde{K}_S$. We bound the adversary $\mathcal{A}$'s difference in advantage by a reduction $\mathcal{B}_3$ to the IND-CCA security of OKEM. $\mathcal{B}_3$ 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 . Next, through four (swap)-PRF hops ($\mathcal{B}_4$ – $\mathcal{B}_7$), we show that this turns *auth* into a random fl_1 -bit value leaving $\mathcal{A}$ with a $1/2^{fl_1}$ 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 $(n_s + n_r q_C)$ 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, π^* , and its partner session, π_p^* (term $\text{Adv}_{\text{OKEM}}^{\text{IND-1CCA}}(\mathcal{B}_8)$). In particular, we replace the ephemeral KEM key K_e with a uniformly random $\tilde{K}_e$ in the target session π^* . All values derived from K_e in π^* use the randomized value $\tilde{K}_e$. We bound the adversary $\mathcal{A}$'s difference in advantage by a reduction $\mathcal{B}_8$ to the IND-1CCA security of OKEM. $\mathcal{B}_8$ obtains the IND-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 IND-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 . Following this, we apply 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 CHALLENGE query to a pre-determined server), we embed a SPR-CCA challenge in the long-term KEM encapsulation of the CHALLENGE sessions (term $\text{Adv}_{\text{OKEM}}^{\text{SPR-CCA}}(\mathcal{B}_{11})$), randomizing both $\widehat{c}_S$ and K_S . Specifically, 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 OKEM.Encap, the initiator samples $\widehat{c}_S \xleftarrow{\$} \{0, 1\}^{cl}$ and $K_S \xleftarrow{\$} \mathcal{K}$ uniformly at random and derives the ciphertext as $c_S \leftarrow \text{DecodeCtxt}(\widehat{c}_S)$. We bound the adversary $\mathcal{A}$'s difference in advantage by a reduction $\mathcal{B}_{11}$ to the SPR-CCA security of OKEM. $\mathcal{B}_{11}$ obtains the SPR-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 p th-created responder v^* committed to by the adversary. When CHALLENGE 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 SPR-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 . Subsequently, 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 $\text{Adv}_{\text{OKEM}}^{\text{ctxt-unif}}(\mathcal{B}_{16})$ and $\text{Adv}_{\text{OKEM}}^{\text{pk-unif}}(\mathcal{B}_{17})$) to randomize $\widehat{c}_e$ and $\widehat{pk}_e$. At this point, the protocol messages in CHALLENGE sessions are uniformly distributed like outputs of the simulator $\mathcal{S}_{\text{pq-obfs}}$, concluding this case and the proof. $\square$

7 Discussion and Further Challenges

7.1 Simulating TLS 1.3

Due to the pervasiveness of TLS [40], circumvention tools often attempt to mimic TLS patterns [25] 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 [40] 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 behavior and distribution of handshake fields; we discuss candidate fields for embedding additional information in the full version [28]. To achieve probing resistance, an obfuscated protocol mimicking TLS 1.3 should copy the same failure modes. Our model currently does not capture responses to probes other than empty messages, but can be easily generalized.

7.2 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 fall into the same class and thus, all handshakes could be proven indistinguishable from another. In reality, protocols have a surfeit of properties that 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 classify 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 that aim 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 can be found in the full version [28].

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 [51]. 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 an effective directionality pattern 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 [29, 34, 49]. 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 [20] to view the protocol as a whole. We note, however, that evidence of real-world blocking [51] 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 [20] include termination of TCP connections into their model of the data transfer phase and suggest that the 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.⁵ Our current model is incapable of capturing this property because we model the handshake as a transcript of messages. Modeling timing behavior is challenging because it requires a streaming-like notion [23] 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.

7.3 Obfuscating Further Post-Quantum KEMs

With Kemeleon for Kyber/ML-KEM, we present a first construction for mapping public keys and ciphertext of a post-quantum KEM to random bytestrings, and also make the required properties formal. Finding such mappings for further post-quantum KEM schemes is an interesting avenue for future work. Strong pseudorandomness under chosen-ciphertext attacks can be a useful building block towards such mappings, as we show for ML-Kemeleon (cf. Lemma 2.15), and has been studied for the NIST PQC Round 3 KEM candidates in [53]. This suggests that, for example, the lattice-based public keys and ciphertexts of FrodoKEM [36] already are uniform strings. In general, finding an appropriate obfuscation mechanism is however non-trivial; for example, an attempt to create an isogeny-based password authenticated key exchange (PAKE) failed due in part to the difficulty of making isogeny-based public keys indistinguishable from random bitstrings [9].

Acknowledgements

We would like to thank those from the anti-censorship research community who compiled many of the resources referenced in Section 7.2. We also thank David Fifield and Lindsey Tulloch for useful discussions on obfs4, Kien Tuong Truong for proposing the name Kemeleon, Stanislaw Jarecki for insightful discussions about covert authenticated key exchange, Michael Rosenberg for discussions on obfuscated KEMs, Jack Wampler for discussing implementation details of Kemeleon, and the CCS reviewers for helpful comments.

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.

⁵This has previously been discussed on the tor-dev mailing list, see <https://lists.torproject.org/pipermail/tor-dev/2017-June/012310.html>.

- [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, Jckson 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] Simurgh Aryan, Homa Aryan, and J. Alex Halderman. Internet censorship in Iran: A first look. In Jedidiah R. Crandall and Joss Wright, editors, *3rd USENIX Workshop on Free and Open Communications on the Internet, FOCI '13*. USENIX Association, 2013.
- [8] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber algorithm specifications and supporting documentation (v3.02), 2021.
- [9] Reza Azarderakhsh, David Jao, Brian Koziel, Jason T. LeGrow, Vladimir Soukharev, and Oleg Tarasikin. How not to create an isogeny-based PAKE. Cryptology ePrint Archive, Report 2020/361, 2020. <https://eprint.iacr.org/2020/361>.
- [10] 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.
- [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.
- [14] Cecylia Bocovich, Arlo Breault, David Fifield, Serene, and Xiaokang Wang. Snowflake, a censorship circumvention system using temporary WebRTC proxies. In *33rd USENIX Security Symposium, USENIX Security 2024*, 2024. To appear.
- [15] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrede 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] Culture Club. Karma Chameleon. In *Colour by Numbers*, 1983.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] 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.
- [24] Sergey Frolov, Jack Wampler, and Eric Wustrow. Detecting probe-resistant proxies. In *NDSS 2020*. The Internet Society, February 2020.
- [25] Sergey Frolov and Eric Wustrow. The use of TLS in censorship circumvention. In *NDSS 2019*. The Internet Society, February 2019.
- [26] Ian Goldberg. [tor-dev] RFC on obfs3 pluggable transport. <https://lists.torproject.org/pipermail/tor-dev/2012-December/004245.html>, 2012.
- [27] Ian Goldberg, Douglas Stebila, and Berkant Ustaoglu. Anonymity and one-way authentication in key exchange protocols. *DCC*, 67(2):245–269, 2013.
- [28] Felix Günther, Douglas Stebila, and Shannon Veitch. Obfuscated key exchange. Cryptology ePrint Archive, Paper 2024/1086, 2024. Full version. <https://eprint.iacr.org/2024/1086>.
- [29] 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.
- [30] 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.
- [31] Stanislaw Jarecki. Practical covert authentication. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of LNCS, pages 611–629. Springer, Heidelberg, March 2014.
- [32] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of LNCS, pages 631–648. Springer, Heidelberg, August 2010.
- [33] A. Langley, M. Hamburg, and S. Turner. Elliptic Curves for Security. RFC 7748 (Informational), January 2016.
- [34] 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.
- [35] 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.
- [36] Michael Naehrig, Erdem Alkim, Joppe Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM. Technical report, National Institute of Standards and Technology, 2020. available at <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>.
- [37] NIST. Module-lattice-based key-encapsulation mechanism standard, August 2024. FIPS 203. <https://doi.org/10.6028/NIST.FIPS.203>.
- [38] Tatsuki 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.
- [39] The Tor Project. lyrebird, version 348eddc8. <https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/lyrebird>, 2023.
- [40] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard), August 2018.
- [41] 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.
- [42] Benjamin Schwartz and Christopher Patton. The pseudorandom extension for cTLS. <https://www.ietf.org/archive/id/draft-cpbs-pseudorandom-ctls-01.html>, 2022.
- [43] 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.
- [44] 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.
- [45] 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.
- [46] The Tor Project. Tor metrics: Bridge users by transport. <https://metrics.torproject.org/userstats-bridge-transport.html?transport=!%3COR%3E&transport=obfs4>, 2024.
- [47] 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.
- [48] 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.
- [49] 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.
- [50] Bas Westerbaan and Cefan Daniel Rubin. Defending against future threats: Cloudflare goes post-quantum. <https://blog.cloudflare.com/post-quantum-for-all>, 2022.
- [51] 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.
- [52] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman. Telex: Anticensorship in the network infrastructure. In *USENIX Security 2011*. USENIX Association, August 2011.
- [53] 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.