Separate Your Domains: NIST PQC KEMs, Oracle Cloning and Read-Only Indifferentiability

Mihir Bellare[1], Hannah Davis[1], and Felix Günther[2]

[1] Dept. of Computer Science & Engineering, University of California San Diego.
{mihir,h3davis}@eng.ucsd.edu https://cseweb.ucsd.edu/~{mihir,h3davis}
[2] Department of Computer Science, ETH Zürich.
mail@felixguenther.info https://www.felixguenther.info

Abstract. It is convenient and common for schemes in the random oracle model to assume access to multiple random oracles (ROs), leaving to implementations the task —we call it oracle cloning— of constructing them from a single RO. The first part of the paper is a case study of oracle cloning in KEM submissions to the NIST Post-Quantum Cryptography standardization process. We give key-recovery attacks on some submissions arising from mistakes in oracle cloning, and find other submissions using oracle cloning methods whose validity is unclear. Motivated by this, the second part of the paper gives a theoretical treatment of oracle cloning. We give a definition of what is an "oracle cloning method" and what it means for such a method to "work," in a framework we call read-only indifferentiability, a simple variant of classical indifferentiability that yields security not only for usage in single-stage games but also in multi-stage ones. We formalize domain separation, and specify and study many oracle cloning methods, including common domain-separating ones, giving some general results to justify (prove read-only indifferentiability of) certain classes of methods. We are not only able to validate the oracle cloning methods used in many of the unbroken NIST PQC KEMs, but also able to specify and validate oracle cloning methods that may be useful beyond that.

1 Introduction

Theoretical works giving, and proving secure, schemes in the random oracle (RO) model [11], often, for convenience, assume access to *multiple, independent* ROs. Implementations, however, like to implement them all via a *single* hash function like SHA256 that is assumed to be a RO.

The transition from one RO to many is, in principle, easy. One can use a method suggested by BR [11] and usually called "domain separation." For example to build three random oracles H_1, H_2, H_3 from a single one, H, define

$$H_1(x) = H(\langle 1\rangle \| x), \quad H_2(x) = H(\langle 2\rangle \| x) \quad \text{and} \quad H_3(x) = H(\langle 3\rangle \| x)\ , \qquad (1)$$

where $\langle i\rangle$ is the representation of integer i as a bit-string of some fixed length, say one byte. One might ask if there is justifying theory: a proof that the above

"works," and a definition of what "works" means. A likely response is that it is obvious it works, and theory would be pedantic.

If it were merely a question of the specific domain-separation method of Equation (1), we'd be inclined to agree. But we have found some good reasons to revisit the question and look into theoretical foundations. They arise from the NIST Post-Quantum Cryptography (PQC) standardization process [35].

We analyzed the KEM submissions. We found attacks, breaking some of them, that arise from incorrect ways of turning one random oracle into many, indicating that the process is error-prone. We found other KEMs where methods other than Equation (1) were used and whether or not they work is unclear. In some submissions, instantiations for multiple ROs were left unspecified. In others, they differed between the specification and reference implementation.

Domain separation as per Equation (1) is a *method*, not a *goal*. We identify and name the underlying goal, calling it *oracle cloning*— given one RO, build many, independent ones. (More generally, given m ROs, build $n > m$ ROs.) We give a definition of what is an "oracle cloning method" and what it means for such a method to "work," in a framework we call read-only indifferentiability, a simple variant of classical indifferentiability [29]. We specify and study many oracle cloning methods, giving some general results to justify (prove read-only indifferentiability of) certain classes of them. The intent is not only to validate as many NIST PQC KEMs as possible (which we do) but to specify and validate methods that will be useful beyond that.

Below we begin by discussing the NIST PQC KEMs and our findings on them, and then turn to our theoretical treatment and results.

NIST PQC KEMs. In late 2016, NIST put out a call for post-quantum cryptographic algorithms [35]. In the first round they received 28 submissions targeting IND-CCA-secure KEMs, of which 17 remain in the second round [37].

Recall that in a KEM (Key Encapsulation Mechanism) KE, the encapsulation algorithm $\mathsf{KE.E}$ takes the public key pk (but no message) to return a symmetric key K and a ciphertext C^* encapsulating it, $(C^*, K) \xleftarrow{\$} \mathsf{KE.E}(pk)$. Given an IND-CCA KEM, one can easily build an IND-CCA PKE scheme by hybrid encryption [18], explaining the focus of standardization on the KEMs.

Most of the KEM submissions (23 in the first round, 15 in the second round) are constructed from a weak (OW-CPA, IND-CPA, ...) PKE scheme using either a method from Hofheinz, Hövelmanns and Kiltz (HHK) [24] or a related method from [21, 40, 27]. This results in a KEM KE_4, the subscript to indicate that it uses up to four ROs that we'll denote H_1, H_2, H_3, H_4. Results of [24, 21, 40, 27] imply that KE_4 is provably IND-CCA, *assuming the ROs* H_1, H_2, H_3, H_4 *are independent.*

Next, the step of interest for us, the oracle cloning: they build the multiple random oracles via a single RO H, replacing H_i with an oracle $\mathbf{F}[H](i, \cdot)$, where we refer to the construction $\mathbf{F}$ as a "cloning functor," and $\mathbf{F}[H]$ means that $\mathbf{F}$ gets oracle access to H. This turns KE_4 into a KEM KE_1 that uses only a *single* RO H, allowing an implementation to instantiate the latter with a single NIST-recommended primitive like SHA3-512 or SHAKE256 [36]. (In some cases, KE_1

uses a number of ROs that is more than one but less than the number used by KE_4, which is still oracle cloning, but we'll ignore this for now.)

Often the oracle cloning method (cloning functor) is not specified in the submission document; we obtained it from the reference implementation. Our concern is the security of this method and the security of the final, single-RO-using KEM KE_1. (As above we assume the starting KE_4 is secure if its four ROs are independent.)

Oracle cloning in submissions. We surveyed the relevant (first- and second-round) NIST PQC KEM submissions, looking in particular at the reference code, to determine what choices of cloning functor $\mathbf{F}$ was made, and how it impacted security of KE_1. Based on our findings, we classify the submissions into groups as follows.

First is a group of *successfully attacked* submissions. We discover and specify attacks, enabled through erroneous RO cloning, on three (first-round) submissions: BIG QUAKE [8], DAGS [7] and Round2 [22]. (Throughout the paper, first-round submissions are in gray, second-round submissions in **bold**.) Our attacks on BIG QUAKE and Round2 recover the symmetric key K from the ciphertext C^* and public key. Our attack on DAGS succeeds in partial key recovery, recovering 192 bits of the symmetric key. These attacks are very fast, taking at most about the same time as taken by the (secret-key equipped, prescribed) decryption algorithm to recover the key. None of our attacks needs access to a decryption oracle, meaning we violate much more than IND-CCA.

Next is submissions with *questionable oracle cloning*. We put just one in this group, namely **NewHope** [2]. Here we do not have proof of security in the ROM for the final instantiated scheme KE_1. We do show that the cloning methods used here do not achieve our formal notion of rd-indiff security, but this does not result in an attack on KE_1, so we do not have a practical attack either. We recommend changes in the cloning methods that permit proofs.

Next is a group of ten submissions that use *ad-hoc oracle cloning* methods — as opposed, say, to conventional domain separation as per Equation (1)— but for which our results (to be discussed below) are able to prove security of the final single-RO scheme. In this group are **BIKE** [3], KCL [44], LAC [28], Lizard [16], LOCKER [4], Odd Manhattan [38], **ROLLO-II** [30], **Round5** [6], **SABER** [19] and Titanium [43]. Still, the security of these oracle cloning methods remains brittle and prone to vulnerabilities under slight changes.

A final group of twelve submissions *did well*, employing something like Equation (1). In particular our results can prove these methods secure. In this group are **Classic McEliece** [13], **CRYSTALS-Kyber** [5], EMBLEM [41], **FrodoKEM** [34], **HQC** [32], LIMA [42], NTRU-HRSS-KEM [25], **NTRU Prime** [14], **NTS-KEM** [1], **RQC** [31], **SIKE** [26] and **ThreeBears** [23].

This classification omits 14 KEM schemes that do not fit the above framework. (For example they do not target IND-CCA KEMs, do not use HHK-style transforms, or do not use multiple random oracles.)

Lessons and response. We see that oracle cloning is error-prone, and that it is sometimes done in ad-hoc ways whose validity is not clear. We suggest that

oracle cloning not be left to implementations. Rather, scheme designers should give proof-validated oracle cloning methods for their schemes. To enable this, we initiate a theoretical treatment of oracle cloning. We formalize oracle cloning methods, define what it means for one to be secure, and specify a library of proven-secure methods from which designers can draw. We are able to justify the oracle cloning methods of many of the unbroken NIST PQC KEMs. The framework of read-only indifferentiability we introduce and use for this purpose may be of independent interest.

The NIST PQC KEMs we break are first-round candidates, not second-round ones, and in some cases other attacks on the same candidates exist, so one may say the breaks are no longer interesting. We suggest reasons they are. Their value is illustrative, showing not only that errors in oracle cloning occur in practice, but that they can be devastating for security. In particular, the extensive and long review process for the first-round NIST PQC submissions seems to have missed these simple attacks, perhaps due to lack of recognition of the importance of good oracle cloning.

Indifferentiability background. Let SS, ES be sets of functions. (We will call them the starting and ending function spaces, respectively.) A functor $\mathbf{F}\colon \mathsf{SS} \to \mathsf{ES}$ is a deterministic algorithm that, given as oracle a function $s \in \mathsf{SS}$, defines a function $\mathbf{F}[s] \in \mathsf{ES}$. Indifferentiability of $\mathbf{F}$ is a way of defining what it means for $\mathbf{F}[s]$ to emulate e when s, e are randomly chosen from SS, ES, respectively. It permits a "composition theorem" saying that if $\mathbf{F}$ is indifferentiable then use of e in a scheme can be securely replaced by use of $\mathbf{F}[s]$.

Maurer, Renner and Holenstein (MRH) [29] gave the first definition of indifferentiability and corresponding composition theorem. However, Ristenpart, Shacham and Shrimpton (RSS) [39] pointed out a limitation, namely that it only applies to single-stage games. MRH-indiff fails to guarantee security in multi-stage games, a setting that includes many goals of interest including security under related-key attack, deterministic public-key encryption and encryption of key-dependent messages. Variants of MRH-indiff [17, 39, 20, 33] tried to address this, with limited success.

Rd-indiff. Indifferentiability is the natural way to treat oracle cloning. A cloning of one function into n functions ($n = 4$ above) can be captured as a functor (we call it a cloning functor) $\mathbf{F}$ that takes the single RO s and for each $i \in [1..n]$ defines a function $\mathbf{F}[s](i, \cdot)$ that is meant to emulate a RO. We will specify many oracle cloning methods in this way.

We define in Section 4 a variant of indifferentiability we call read-only indifferentiability (rd-indiff). The simulator —unlike for reset-indiff [39]— has access to a game-maintained state st, but —unlike MRH-indiff [29]— that state is read-only, meaning the simulator cannot alter it across invocations. Rd-indiff is a stronger requirement than MRH-indiff (if $\mathbf{F}$ is rd-indiff then it is MRH-indiff) but a weaker one than reset-indiff (if $\mathbf{F}$ is reset-indiff then it is rd-indiff). Despite the latter, rd-indiff, like reset-indiff, admits a composition theorem showing that an rd-indiff $\mathbf{F}$ may securely substitute a RO even in multi-stage games. (The proof of RSS [39] for reset-indiff extends to show this.) We do not use reset-

indiff because some of our cloning functors do not meet it, but they do meet rd-indiff, and the composition benefit is preserved.

General results. In Section 4, we define *translating* functors. These are simply ones whose oracle queries are non-adaptive. (In more detail, a translating functor determines from its input W a list of queries, makes them to its oracle and, from the responses and W, determines its output.) We then define a condition on a translating functor $\mathbf{F}$ that we call *invertibility* and show that if $\mathbf{F}$ is an invertible translating functor then it is rd-indiff. This is done in two parts, Theorems 1 and 2, that differ in the degree of invertibility assumed. The first, assuming the greater degree of invertibility, allows a simpler proof with a simulator that does not need the read-only state allowed in rd-indiff. The second, assuming the lesser degree of invertibility, depends on a simulator that makes crucial use of the read-only state. It sets the latter to a key for a PRF that is then used to answer queries that fall outside the set of ones that can be trivially answered under the invertibility condition. This use of a computational primitive (a PRF) in the indifferentiability context may be novel and may seem odd, but it works.

We apply this framework to analyze particular, practical cloning functors, showing that these are translating and invertible, and then deducing their rd-indiff security. But the above-mentioned results are stronger and more general than we need for the application to oracle cloning. The intent is to enable further, future applications.

Analysis of oracle cloning methods. We formalize oracle cloning as the task of designing a functor (we call it a cloning functor) $\mathbf{F}$ that takes as oracle a function $s \in \mathsf{SS}$ in the starting space and returns a two-input function $e = \mathbf{F}[s] \in \mathsf{ES}$, where $e(i, \cdot)$ represents the i-th RO for $i \in [1..n]$. Section 5 presents the cloning functors corresponding to some popular and practical oracle cloning methods (in particular ones used in the NIST PQC KEMs), and shows that they are translating and invertible. Our above-mentioned results allow us to then deduce they are rd-indiff, which means they are safe to use in most applications, even ones involving multi-stage games. This gives formal justification for some common oracle cloning methods. We now discuss some specific cloning functors that we treat in this way.

The prefix (cloning) functor $\mathbf{F}_{\mathrm{pf}(\mathbf{p})}$ is parameterized by a fixed, public vector $\mathbf{p}$ such that no entry of $\mathbf{p}$ is a prefix of any other entry of $\mathbf{p}$. Receiving function s as an oracle, it defines function $e = \mathbf{F}_{\mathrm{pf}(\mathbf{p})}[s]$ by $e(i, X) = s(\mathbf{p}[i] \| X)$, where $\mathbf{p}[i]$ is the i^{th} element of vector $\mathbf{p}$. When $\mathbf{p}[i]$ is a fixed-length bitstring representing the integer i, this formalizes Equation (1).

Some NIST PQC submissions use a method we call output splitting. The simplest case is that we want $e(i, \cdot), \ldots, \epsilon(n, \cdot)$ to all have the same output length L. We then define $e(i, X)$ as bits $(i-1)L+1$ through iL of the given function s applied to X. That is, receiving function s as an oracle, the splitting (cloning) functor $\mathbf{F}_{\mathrm{spl}}$ returns function $e = \mathbf{F}_{\mathrm{spl}}[s]$ defined by $e(i, X) = s(X)[(i-1)L+1..iL]$.

An interesting case, present in some NIST PQC submissions, is trivial cloning: just set $e(i, X) = s(X)$ for all X. We formalize this as the identity (cloning) functor $\mathbf{F}_{\mathrm{id}}$ defined by $\mathbf{F}_{\mathrm{id}}[s](i, X) = s(X)$. Clearly, this is not always secure. It can

be secure, however, for usages that restrict queries in some way. One such restriction, used in several NIST PQC KEMs, is length differentiation: $e(i, \cdot)$ is queried only on inputs of some length l_i, where $l_1, \ldots, l_n$ are chosen to be distinct. We are able to treat this in our framework using the concept of working domains that we discuss next, but we warn that this method is brittle and prone to misuse.

Working domains. One could capture trivial cloning with length differentiation as a restriction on the domains of the ending functions, but this seems artificial and dangerous because the implementations do not enforce any such restriction; the functions there are defined on their full domains and it is, apparently, left up to applications to use the functions in a way that does not get them into trouble. The approach we take is to leave the functions defined on their full domains, but define and ask for security over a subdomain, which we called the working domain. A choice of working domain $\mathcal{W}$ accordingly parameterizes our definition of rd-indiff for a functor, and also the definition of invertibility of a translating functor. Our result says that the identity functor is rd-indiff for certain choices of working domains that include the length differentiation one.

Making the working domain explicit will, hopefully, force the application designer to think about, and specify, what it is, increasing the possibility of staying out of trouble. Working domains also provide flexibility and versatility under which different applications can make different choices of the domain.

Working domains not being present in prior indifferentiability formalizations, the comparisons, above, of rd-indiff with these prior formalizations assume the working domain is the full domain of the ending functions. Working domains alter the comparison picture; a cloning functor which is rd-indiff on a working domain may not be even MRH-indiff on its full domain.

Application to KEMs. The framework above is broad, staying in the land of ROs and not speaking of the usage of these ROs in any particular cryptographic primitive or scheme. As such, it can be applied to analyze RO instantiation in many primitives and schemes. In the full version of this paper [10], we exemplify its application in the realm of KEMs as the target of the NIST PQC designs.

This may seem redundant, since an indifferentiability composition theorem says exactly that once indifferentiability of a functor has been shown, "all" uses of it are secure. However, prior indifferentiability frameworks do not consider working domains, so the known composition theorems apply only when the working domain is the full one. (Thus the reset-indiff composition theorem of [39] extends to rd-indiff so that we have security for applications whose security definitions are underlain by either single or multi-stage games, but only for full working domains.)

To give a composition theorem that is conscious of working domains, we must first ask what they are, or mean, in the application. We give a definition of the *working domain of a KEM* KE. This is the set of all points that the scheme algorithms query to the ending functions in usage, captured by a certain game we give. (Queries of the adversary may fall outside the working domain.) Then we give a working-domain-conscious composition theorem for KEMs that says

the following. Say we are given an IND-CCA KEM KE whose oracles are drawn from a function space $\mathsf{KE.FS}$. Let $\mathbf{F}\colon \mathsf{SS} \to \mathsf{KE.FS}$ be a functor, and let $\overline{\mathsf{KE}}$ be the KEM obtained by implementing the oracles of the KE via $\mathbf{F}$. (So the oracles of this second KEM are drawn from the function space $\overline{\mathsf{KE}}.\mathsf{FS} = \mathsf{SS}$.) Let $\mathcal{W}$ be the working domain of KE, and assume $\mathbf{F}$ is rd-indiff over $\mathcal{W}$. Then $\overline{\mathsf{KE}}$ is also IND-CCA. Combining this with our rd-indiff results on particular cloning functors justifies not only conventional domain separation as an instantiation technique for KEMs, but also more broadly the instantiations in some NIST PQC submissions that do not use domain separation, yet whose cloning functors are rd-diff over the working domain of their KEMs. The most important example is the identity cloning functor used with length differentiation.

A key definitional element of our treatment that allows the above is, following [9], to embellish the *syntax* of a scheme (here a KEM KE) by having it name a function space $\mathsf{KE.FS}$ from which it wants its oracles drawn. Thus, the scheme specification must say how many ROs it wants, and of what domains and ranges. In contrast, in the formal version of the ROM in [11], there is a single, scheme-independent RO that has some fixed domain and range, for example mapping $\{0,1\}^*$ to $\{0,1\}$. This leaves a gap, between the object a scheme wants and what the model provides, that can lead to error. We suggest that, to reduce such errors, schemes specified in standards include a specification of their function space.

2 Oracle Cloning in NIST PQC Candidates

Notation. A KEM scheme KE specifies an encapsulation $\mathsf{KE.E}$ that, on input a public encryption key pk returns a session key K, and a ciphertext C^* encapsulating it, written $(C^*, K) \leftarrow\!\!\$\ \mathsf{KE.E}(pk)$. A PKE scheme PKE specifies an encryption algorithm $\mathsf{PKE.E}$ that, on input pk, message $M \in \{0,1\}^{\mathsf{PKE.ml}}$ and randomness R, deterministically returns ciphertext $C \leftarrow \mathsf{PKE.E}(pk, M; R)$. For neither primitive will we, in this section, be concerned with the key generation or decapsulation / decryption algorithm. We might write $\mathsf{KE}[X_1, X_2, \ldots]$ to indicate that the scheme has oracle access to functions $X_1, X_2, \ldots$, and correspondingly then write $\mathsf{KE.E}[X_1, X_2, \ldots]$, and similarly for PKE.

2.1 Design process

The literature [24, 21, 40, 27] provides many transforms that take a public-key encryption scheme PKE, assumed to meet some weaker-than-IND-CCA notion of security we denote $\mathrm{S}_{\mathrm{pke}}$ (for example, OW-CPA, OW-PCA or IND-CPA), and, with the aid of some number of random oracles, turn PKE into a KEM that is guaranteed (proven) to be IND-CCA *assuming the ROs are independent.* We'll refer to such transforms as *sound*. Many (most) KEMs submitted to the NIST Post-Quantum Cryptography standardization process were accordingly designed as follows:

(1) First, they specify a $\mathrm{S}_{\mathrm{pke}}$-secure public-key encryption scheme PKE.

(2) Second, they pick a sound transform $\mathbf{T}$ and obtain KEM $\mathsf{KE}_4[H_1, H_2, H_3, H_4] = \mathbf{T}[\mathsf{PKE}, H_2, H_3, H_4]$. (The notation is from [24]. The transforms use up to three random oracles that we are denoting H_2, H_3, H_4, reserving H_1 for possible use by the PKE scheme.) We refer to KE_4 (the subscript refers to its using 4 oracles) as the *base* KEM, and, as we will see, it differs across the transforms.

(3) Finally —the under-the-radar step that is our concern— the ROs $H_1, \ldots, H_4$ are constructed from cryptographic hash functions to yield what we call the *final* KEM KE_1. In more detail, the submissions make various choices of cryptographic hash functions $F_1, \ldots, F_m$ that we call the *base functions*, and, for $i = 1, 2, 3, 4$, specify constructions $\mathbf{C}_i$ that, with oracle access to the base functions, define the H_i, which we write as $H_i \leftarrow \mathbf{C}_i[F_1, \ldots, F_m]$. We call this process oracle cloning, and we call H_i the *final functions*. (Common values of m are $1, 2$.) The actual, submitted KEM KE_1 (the subscript because m is usually 1) uses the final functions, so that its encapsulation algorithm can be written as:

$\underline{\mathsf{KE}_1.\mathsf{E}[F_1, \ldots, F_m](pk)}$
For $i = 1, 2, 3, 4$ do $H_i \leftarrow \mathbf{C}_i[F_1, \ldots, F_m]$
$(C^*, K) \leftarrow\!\!\$\ \mathsf{KE}_4.\mathsf{E}[H_1, H_2, H_3, H_4](pk)$
Return (C^*, K)

The question now is whether the final KE_1 is secure. We will show that, for some submissions, it is not. This is true for the choices of base functions $F_1, \ldots, F_m$ made in the submission, but also if these are assumed to be ROs. It is true despite the soundness of the transform, meaning insecurity arises from poor oracle cloning, meaning choices of the constructions $\mathbf{C}_i$. We will then consider submissions for which we have not found an attack. In the latter analysis, we are willing to assume (as the submissions implicitly do) that $F_1, \ldots, F_m$ are ROs, and we then ask whether the final functions are "close" to independent ROs.

2.2 The base KEM

We need first to specify the base KE_4 (the result of the sound transform, from step (2) above). The NIST PQC submissions typically cite one of HHK [24], Dent [21], SXY [40] or JZCWM [27] for the sound transform they use, but our examinations show that the submissions have embellished, combined or modified the original transforms. The changes do *not* (to best of our knowledge) violate soundness (meaning the used transforms still yield an IND-CCA KE_4 if H_2, H_3, H_4 are independent ROs and PKE is $\mathrm{S}_{\mathrm{pke}}$-secure) but they make a succinct exposition challenging. We address this with a framework to unify the designs via a single, but parameterized, transform, capturing the submission transforms by different parameter choices.

Figure 1 (top) shows the encapsulation algorithm $\mathsf{KE}_4.\mathsf{E}$ of the KEM that our parameterized transform associates to PKE and H_1, H_2, H_3, H_4. The parameters are the variables X, Y, Z (they will be functions of other quantities in the

Algorithm $\mathsf{KE}_4.\mathsf{E}[H_1, H_2, H_3, H_4](pk)$:

1 $M \leftarrow\!\!\$\ \{0,1\}^{\mathsf{PKE.ml}}$; $R \leftarrow \varepsilon$
2 If $(\mathsf{D} = \mathsf{true})$ then $R \,\|\, K' \leftarrow H_2(X)$ // $|K'| = \mathsf{k}^*$
3 $C \leftarrow \mathsf{PKE.E}[H_1](pk, M; R)$
4 $C^* \leftarrow C \,\|\, Y$
5 $K \leftarrow H_4(Z)$; Return (C^*, K)

	D	k*	X	Y	Z	Used in
$\mathbf{T}_1$	true	0	M	ε	M	LIMA, Odd Manhattan
$\mathbf{T}_2$	true	0	$pk\|M$	ε	$pk\|M$	**ThreeBears**
$\mathbf{T}_3$	true	0	M	ε	$M\|C$	**BIKE-1-CCA** **BIKE-3-CCA**, **LAC**
$\mathbf{T}_4$	true	0	$M\|pk$	ε	$M\|C$	**SIKE**
$\mathbf{T}_5$	true	0	M	$H_3(X)$	$M\|C$	**HQC**, **RQC**, Titanium
$\mathbf{T}_6$	true	>0	$M\|H_3(pk)$	ε	$K'\|C$	**SABER**
$\mathbf{T}_7$	true	>0	$H_3(pk)\|H_3(M)$	ε	$K'\|H_3(C)$	**CRYSTALS-Kyber**
$\mathbf{T}_8$	true	0	M	$H_3(X)$	M	DAGS, NTRU-HRSS-KEM
$\mathbf{T}_9$	true	0	M	$H_3(X)$	$M\|C\|Y$	**ROLLO-II**, EMBLEM, Lizard, LOCKER, BIG QUAKE
$\mathbf{T}_{10}$	true	>0	$H_4(M)\|H_4(pk)$	$H_3(X)$	$K'\|H_4(C\|Y)$	**NewHope**
$\mathbf{T}_{11}$	true	>0	$M\|pk$	$H_3(X)$	$K'\|C\|Y$	FrodoKEM, Round2 **Round5**
$\mathbf{T}_{12}$	true	>0	$pk\|M$	$H_3(X)$	$K'\|C$	KCL
$\mathbf{T}_{13}$	true	>0	$H_3(pk)\|M$	ε	$C\|K'$	**FrodoKEM**
$\mathbf{T}_{14}$	false	0	$\perp$	$H_3(M)$	$M\|C\|Y$	**Classic McEliece**
$\mathbf{T}_{15}$	true	0	M	ε	$R\|M$	**NTS-KEM**
$\mathbf{T}_{16}$	false	0	$\perp$	$M\|pk$	$M\|C\|Y$	**Streamlined NTRU Prime**
$\mathbf{T}_{17}$	true	0	M	$M\|pk$	$M\|C\|Y$	**NTRU LPRime**

Fig. 1. Top: Encapsulation algorithm of the base KEM scheme produced by our parameterized transform. **Bottom:** Choices of parameters $X, Y, Z, \mathsf{D}, \mathsf{k}^*$ resulting in specific transforms used by the NIST PQC submissions. Second-round submissions are in **bold**, first-round submissions in gray. Submissions using different transforms in the two rounds appear twice.

algorithms), a boolean D, and an integer k^*. When choices of these are made, one gets a fully-specified transform and corresponding base KEM KE_4. Each row in the table in the same Figure shows one such choice of parameters, resulting in 15 fully-specified transforms. The final column shows the submissions that use the transform.

The encapsulation algorithm at the top of Figure 1 takes input a public key pk and has oracle access to functions H_1, H_2, H_3, H_4. At line 1, it picks a random seed M of length the message length of the given PKE scheme. Boolean D being true (as it is with just one exception) means $\mathsf{PKE.E}$ is randomized. In that case, line 2 applies H_2 to X (the latter, determined as per the table, depends on M and possibly also on pk) and parses the output to get coins R for $\mathsf{PKE.E}$ and possibly (if the parameter $\mathsf{k}^* \neq 0$) an additional string K'. At line 3, a ciphertext C is produced by encrypting the seed M using $\mathsf{PKE.E}$ with public key pk and coins R. In some schemes, a second portion of the ciphertext, Y, often called the "confirmation", is derived from X or M, using H_3, as shown in the table, and line 4 then defines C^*. Finally, H_4 is used as a key derivation function to extract a symmetric key K from the parameter Z, which varies widely among transforms.

In total, 26 of the 39 NIST PQC submissions which target KEMs in either the first or second round use transforms which fall into our framework. The remaining schemes do not use more than one random oracle, construct KEMs without transforming PKE schemes, or target security definitions other than IND-CCA.

2.3 Submissions we break

We present attacks on `BIG QUAKE` [8], `DAGS` [7], and `Round2` [22]. These attacks succeed in full or partial recovery of the encapsulated KEM key from a ciphertext, and are extremely fast. We have implemented the attacks to verify them.

Although none of these schemes progressed to Round 2 of the competition without significant modification, to the best of our knowledge, none of the attacks we described were pointed out during the review process. Given the attacks' superficiality, this is surprising and suggests to us that more attention should be paid to oracle cloning methods and their vulnerabilities during review.

Randomness-based decryption. The PKE schemes used by `BIG QUAKE` and `Round2` have the property that given a ciphertext $C \leftarrow \mathsf{PKE.E}(pk, M; R)$ and also given the coins R, it is easy to recover M, even without knowledge of the secret key. We formalize this property, saying PKE allows randomness-based decryption, if there is an (efficient) algorithm $\mathsf{PKE.DecR}$ such that $\mathsf{PKE.DecR}(pk, \mathsf{PKE.E}(pk, M; R), R) = M$ for any public key pk, coins R and message m. This will be used in our attacks.

Attack on `BIG QUAKE`. The base KEM $\mathsf{KE}_1[H_1, H_2, H_3, H_4]$ is given by the transform $\mathbf{T}_9$ in the table of Figure 1. The final KEM $\mathsf{KE}_2[F]$ uses a single function F to instantiate the random oracles, which it does as follows. It sets $H_3 = H_4 = F$ and $H_2 = W[F] \circ F$ for a certain function W (the rejection sampling algorithm) whose details will not matter for us. The notation $W[F]$ meaning that W has oracle access to F. The following attack (explanations after the pseudocode) recovers the encapsulated KEM key K from ciphertext $C^* \leftarrow\!\!\$\ \mathsf{KE}_1.\mathsf{E}[F](pk)$—

Adversary $\mathcal{A}[F](pk, C^*)$ // Input public key and ciphertext, oracle for F
1. $C\|Y \leftarrow C^*$ // Parse C^* to get PKE ciphertext C and $Y = H_3(M)$
2. $R \leftarrow W[F](Y)$ // Apply function $W[F]$ to Y to recover coins R
3. $M \leftarrow \mathsf{PKE.DecR}(pk, C, R)$ // Use randomness-based decryption for PKE
4. $K \leftarrow F(M)$; Return K

As per $\mathbf{T}_9$ we have $Y = H_3(M) = F(M)$. The coins for $\mathsf{PKE.E}$ are $R = H_2(M) = (W[F] \circ F)(M) = W[F](F(M)) = W[F](Y)$. Since Y is in the ciphertext, the coins R can be recovered as shown at line 2. The PKE scheme allows randomness-based decryption, so at line 3 we can recover the message M underlying C using algorithm $\mathsf{PKE.DecR}$. But $K = H_4(M) = F(M)$, so K can now be recovered as well. In conclusion, the specific cloning method chosen by `BIG QUAKE` leads to complete recovery of the encapsulated key from the ciphertext.

Attack on `Round2`. The base KEM $\mathsf{KE}_1[H_2, H_3, H_4]$ is given by the transform $\mathbf{T}_{11}$ in the table of Figure 1. The final KEM $\mathsf{KE}_2[F]$ uses a single base function F to instantiate the final functions, which it does as follows. It sets $H_4 = F$. The specification and reference implementation differ in how H_2, H_3 are defined: In the former, $H_2(x) = F(F(x)) \,\|\, F(x)$ and $H_3(x) = F(F(F(x)))$, while, in the latter, $H_2(x) = F(F(F(x))) \,\|\, F(x)$ and $H_3(x) = F(F(X))$. These differences arise from differences in the way the output of a certain function $W[F]$ is parsed.

Our attack is on the reference-implementation version of the scheme. We need to also know that the scheme sets k^* so that $R\|K' \leftarrow H_2(X)$ with $H_2(X) = F(F(F(X)))\|F(X)$ results in $R = F(F(F(X)))$. But $Y = H_3(X) = F(F(X))$, so $R = F(Y)$ can be recovered from the ciphertext. Again exploiting the fact that the PKE scheme allows randomness-based decryption, we obtain the following attack that recovers the encapsulated KEM key K from ciphertext $C^* \leftarrow\!\!\$\, \mathsf{KE}_1.\mathsf{E}[F](pk)$—

Adversary $\mathcal{A}[F](pk, C^*)$ // Input public key and ciphertext, oracle for F
1. $C\|Y \leftarrow C^*$; $R \leftarrow F(Y)$
2. $M \leftarrow \mathsf{PKE.DecR}(pk, C, R)$; $K \leftarrow F(M)$; Return K

This attack exploits the difference between the way H_2, H_3 are defined across the specification and implementation, which may be a bug in the implementation with regard to the parsing of $W[F](x)$. However, the attack also exploits dependencies between H_2 and H_3, which ought not to exist when instantiating what are required to be distinct random oracles.

`Round2` was incorporated into the second-round submission **`Round5`**, which specifies a different base function and cloning functor (the latter of which uses the secure method we call "output splitting") to instantiate oracles H_2 and H_3. This attack therefore does not apply to **`Round5`**.

Attack on DAGS. If x is a byte string we let $x[i]$ be its i-th byte, and if x is a bit string we let x_i be its i-th bit. We say that a function V is an extendable output function if it takes input a string x and an integer ℓ to return an ℓ-byte output, and $\ell_1 \le \ell_2$ implies that $V(x, \ell_1)$ is a prefix of $V(x, \ell_2)$. If $v = v_1v_2v_3v_4v_5v_6v_7v_8$ is a byte then let $Z(v) = 00v_3v_4v_5v_6v_7v_8$ be obtained by zeroing out the first

two bits. If y is a string of ℓ bytes then let $Z'(y) = Z(y[1]) \| \cdots \| Z(y[\ell])$. Now let $V'(x, \ell) = Z'(V(x, \ell))$.

The base KEM $\mathsf{KE}_1[H_1, H_2, H_3, H_4]$ is given by the transform $\mathbf{T}_8$ in the table of Figure 1. The final KEM $\mathsf{KE}_2[V]$ uses an extendable output function V to instantiate the random oracles, which it does as follows. It sets $H_2(x) = V'(x, 512)$ and $H_3(x) = V'(x, 32)$. It sets $H_4(x) = V(x, 64)$.

As per $\mathbf{T}_8$ we have $K = H_4(M)$ and $Y = H_3(M)$. Let L be the first 32 bytes of the 64-byte K. Then $Y = Z'(L)$. So Y reveals $32 \cdot 6 = 192$ bits of K. Since Y is in the ciphertext, this results in a partial encapsulated-key recovery attack. The attack reduces the effective length of K from $64 \cdot 8 = 512$ bits to $512 - 192 = 320$ bits, meaning 37.5% of the encapsulated key is recovered. Also $R = H_2(M)$, so Y, as part of the ciphertext, reveals 32 bytes of R, which does not seem desirable, even though it is not clear how to exploit it for an attack.

2.4 Submissions with unclear security

For the scheme **NewHope** [2], we can give neither an attack nor a proof of security. However, we can show that the final functions H_2, H_3, H_4 produced by the cloning functor $\mathbf{F}_{\texttt{NewHope}}$ with oracle access to a single extendable-output function V are differentiable from independent random oracles. The cloning functor $\mathbf{F}_{\texttt{NewHope}}$ sets $H_1(x) = V(x, 128)$ and $H_4 = V(x, 32)$. It computes H_2 and H_3 from V using the output splitting cloning functor. Concretely, KE_2 parses $V(x, 96)$ as $H_2(x) \| H_3(x)$, where H_2 has output length 64 bytes and H_3 has output length 32 bytes. Because V is an extendable-output function, $H_4(x)$ will be a prefix of $H_2(x)$ for any string x.

We do not know how to exploit this correlation to attack the IND-CCA security of the final KEM scheme $\mathsf{KE}_2[V]$, and we conjecture that, due to the structure of $\mathbf{T}_{10}$, no efficient attack exists. We can, however, attack the rd-indiff security of functor $\mathbf{F}_{\texttt{NewHope}}$, showing that that the security proof for the base KEM $\mathsf{KE}_1[H_2, H_3, H_4]$ does not naturally transfer to $\mathsf{KE}_2[V]$. Therefore, in order to generically extend the provable security results for KE_1 to KE_2, it seems advisable to instead apply appropriate oracle cloning methods.

2.5 Submissions with provable security but ambiguous specification

In their reference implementations, these submissions use cloning functors which we can and do validate via our framework, providing provable security in the random oracle model for the final KEM schemes. However, the submission documents do not clearly specify a secure cloning functor, meaning that variant implementations or adaptations may unknowingly introduce weaknesses. The schemes **BIKE** [3], KCL [44], LAC [28], Lizard [16], LOCKER [4], Odd Manhattan [38], **ROLLO-II** [30], **Round5** [6], **SABER** [19] and Titanium [43] fall into this group.

Length differentiation. Many of these schemes use the "identity" functor in their reference implementations, meaning that they set the final functions $H_1 = H_2 = H_3 = H_4 = F$ for a single base function F. If the scheme

$\mathsf{KE}_1[H_1, H_2, H_3, H_4]$ never queries two different oracles on inputs of a single length, the domains of $H_1, \ldots, H_4$ are implicitly separated. Reference implementations typically enforce this separation by fixing the input length of every call to F. Our formalism calls this query restriction "length differentiation" and proves its security as an oracle cloning method. We also generalize it to all methods which prevent the scheme from querying any two distinct random oracles on a single input.

In the following, we discuss two schemes from the group, `ROLLO-II` and `Lizard`, where ambiguity about cloning methods between the specification and reference implementation jeopardizes the security of applications using these schemes. It will be important that, like `BIG QUAKE` and `RoundTwo`, the PKE schemes defined by `ROLLO-II` and `Lizard` allow randomness-based decryption.

The scheme `ROLLO-II` [30] defines its base KEM $\mathsf{KE}_1[H_1, H_2, H_3, H_4]$ using the $\mathbf{T}_9$ transform from Figure 1. The submission document states that H_1, H_2, H_3, and H_4 are "typically" instantiated with a single fixed-length hash function F, but does not describe the cloning functors used to do so. If the identity functor is used, so that $H_1 = H_2 = H_3 = H_4 = F$, (or more generally, any functor that sets $H_2 = H_3$), an attack is possible. In the transform $\mathbf{T}_9$, both H_2 and H_3 are queried on the same input M. Then $Y = H_3(M) = F(M) = H_2(M) = R$ leaks the PKE's random coins, so the following attack will allow total key recovery via the randomness-based decryption.

Adversary $\mathcal{A}[F](pk, C^*)$ // Input public key and ciphertext, oracle for F
1. $C \| Y \leftarrow C^*$; $M \leftarrow \mathsf{PKE.DecR}(pk, C, Y)$ // ($Y = R$ is the coins)
2. $K \leftarrow F(M \,\|\, C \,\|\, Y)$; Return K

In the reference implementation of `ROLLO-II`, however, H_2 is instantiated using a second, independent function V instead of F, which prevents the above attack. Although the random oracles H_1, H_3 and H_4 are instantiated using the identity functor, they are never queried on the same input thanks to length differentiation. As a result, the reference implementation of `ROLLO-II` is provably secure, though alternate implementations could be both compliant with the submission document and completely insecure. The relevant portions of both the specification and the reference implementation were originally found in the corresponding first-round submission (`LOCKER`).

`Lizard` [16] also follows transform $\mathbf{T}_9$ to produce its base KEM $\mathsf{KE}_1[H_2, H_3, H_4]$. Its submission document suggests instantiation with a single function F as follows: it sets $H_3 = H_4 = F$, and it sets $H_2 = W \circ F$ for some postprocessing function W whose details are irrelevant here. Since, in $\mathbf{T}_9$, $Y = H_3(M) = F(M)$ and $R = H_2(M) = W \circ F(M) = W(Y)$, the randomness R will again be leaked through Y in the ciphertext, permitting a key-recovery attack using randomness-based decryption much like the others we have described. This attack is prevented in the reference implementation of `Lizard`, which instantiates H_3 and H_4 using an independent function G. The domains of H_3 and H_4 are separated by length differentiation. This allows us to prove the security of the final KEM $\mathsf{KE}_2[G, F]$, as defined by the reference implementation.

However, the length differentiation of H_3 and H_4 breaks down in the chosen-ciphertext-secure PKE variant specification of `Lizard`, which transforms KE_1. The PKE scheme, given a plaintext M, computes $R = H_2(M)$ and $Y = H_3(M)$ according to $\mathbf{T}_9$, but it computes $K = H_4(M)$, then includes the value $B = K \oplus M$ as part of the ciphertext C^*. Both the identity functor and the functor used by the KEM reference implementation set $H_3 = H_4$, so the following attack will extract the plaintext from any ciphertext–

Adversary $\mathcal{A}(pk, C^*)$ // Input public key and ciphertext
1. $C \| B \| Y \leftarrow C^*$ // Parse C^* to get Y and $B = M \oplus K$
2. $M \leftarrow Y \oplus B$; Return M // $Y = H_3(M) = H_4(M) = K$ is the mask.

The reference implementation of the public-key encryption schemes prevents the attack by cloning H_3 and H_4 from G via a third cloning functor, this one using the output splitting method. Yet, the inconsistency in the choice of cloning functors between the specification and both implementations underlines that ad-hoc cloning functors may easily "get lost" in modifications or adaptations of a scheme.

2.6 Submissions with clear provable security

Here we place schemes which explicitly discuss their methods for domain separation and follow good practice in their implementations: **`Classic McEliece`** [13], **`CRYSTALS-Kyber`** [5], `EMBLEM` [41], **`FrodoKEM`** [34], **`HQC`** [32], `LIMA` [42], `NTRU-HRSS-KEM` [25], **`NTRU Prime`** [14], **`NTS-KEM`** [1], **`RQC`** [31], **`SIKE`** [26] and **`ThreeBears`** [23]. These schemes are careful to account for dependencies between random oracles that are considered to be independent in their security models. When choosing to clone multiple random oracles from a single primitive, the schemes in this group use padding bytes, deploy hash functions designed to accommodate domain separation, or restrictions on the length of the inputs which are codified in the specification. These explicit domain separation techniques can be cast in the formalism we develop in this work.

`HQC` and **`RQC`** are unique among the PQC KEM schemes in that their specifications warn that the identity functor admits key-recovery attacks. As protection, they recommend that H_2 and H_3 be instantiated with unrelated primitives.

Signatures. Although the main focus of this paper is on domain separation in KEMs, we wish to note that these issues are not unique to KEMs. At least one digital signature scheme in the second round of the NIST PQC competition, **`MQDSS`** [15], models multiple hash functions as independent random oracles in its security proof, then clones them from the same primitive without explicit domain separation. We have not analyzed the NIST PQC digital signature schemes' security to see whether more subtle domain separation is present, or whether oracle collisions admit the same vulnerabilities to signature forgery as they do to session key recovery. This does, however, highlight that the problem of random oracle cloning is pervasive among more types of cryptographic schemes.

3 Preliminaries

BASIC NOTATION. By $[i..j]$ we abbreviate the set $\{i, \ldots, j\}$, for integers $i \leq j$. If $\mathbf{x}$ is a vector then $|\mathbf{x}|$ is its length (the number of its coordinates), $\mathbf{x}[i]$ is its i-th coordinate and $[\mathbf{x}] = \{\mathbf{x}[i] : i \in [1..|\mathbf{x}|]\}$ is the set of its coordinates. The empty vector is denoted $()$. If S is a set, then S^* is the set of vectors over S, meaning the set of vectors of any (finite) length with coordinates in S. Strings are identified with vectors over $\{0,1\}$, so that if $x \in \{0,1\}^*$ is a string then $|x|$ is its length, $x[i]$ is its i-th bit, and $x[i..j]$ is the substring from its i-th to its j-th bit (including), for $i \leq j$. The empty string is ε. If x, y are strings then we write $x \preceq y$ to indicate that x is a prefix of y. If S is a finite set then $|S|$ is its size (cardinality). A set $S \subseteq \{0,1\}^*$ is *length closed* if $\{0,1\}^{|x|} \subseteq S$ for all $x \in S$.

We let $y \leftarrow A[\mathsf{O}_1, \ldots](x_1, \ldots; r)$ denote executing algorithm A on inputs $x_1, \ldots$ and coins r, with access to oracles $\mathsf{O}_1, \ldots$, and letting y be the result. We let $y \leftarrow\!\!\$\, A[\mathsf{O}_1, \ldots](x_1, \ldots)$ be the resulting of picking r at random and letting $y \leftarrow A[\mathsf{O}_1, \ldots](x_1, \ldots; r)$. We let $\mathrm{OUT}(A[\mathsf{O}_1, \ldots](x_1, \ldots))$ denote the set of all possible outputs of algorithm A when invoked with inputs $x_1, \ldots$ and access to oracles $\mathsf{O}_1, \ldots$. Algorithms are randomized unless otherwise indicated. Running time is worst case. An adversary is an algorithm.

We use the code-based game-playing framework of [12]. A game G (see Figure 2 for an example) starts with an INIT procedure, followed by a non-negative number of additional procedures, and ends with a FIN procedure. Procedures are also called oracles. Execution of adversary $\mathcal{A}$ with game G consists of running $\mathcal{A}$ with oracle access to the game procedures, with the restrictions that $\mathcal{A}$'s first call must be to INIT, its last call must be to FIN, and it can call these two procedures at most once. The output of the execution is the output of FIN. We write $\Pr[\mathrm{G}(\mathcal{A})]$ to denote the probability that the execution of game G with adversary $\mathcal{A}$ results in the output being the boolean true. Note that our adversaries have no output. The role of what in other treatments is the adversary output is, for us, played by the query to FIN. We adopt the convention that the running time of an adversary is the worst-case time to execute the game with the adversary, so the time taken by game procedures (oracles) to respond to queries is included.

FUNCTIONS. As usual $g\colon\ \mathcal{D} \to \mathcal{R}$ indicates that g is a function taking inputs in the domain set $\mathcal{D}$ and returning outputs in the range set $\mathcal{R}$. We may denote these sets by $\mathrm{Dom}(g)$ and $\mathrm{Rng}(g)$, respectively.

We say that $g\colon\ \mathrm{Dom}(g) \to \mathrm{Rng}(g)$ has output length ℓ if $\mathrm{Rng}(g) = \{0,1\}^\ell$. We say that g is a single output-length (sol) function if there is some ℓ such that g has output length ℓ and also the set $\mathcal{D}$ is length closed. We let $\mathrm{SOL}(\mathcal{D}, \ell)$ denote the set of all sol functions $g\colon\ \mathcal{D} \to \{0,1\}^\ell$.

We say g is an extendable output length (xol) function if the following are true: (1) $\mathrm{Rng}(g) = \{0,1\}^*$ (2) there is a length-closed set $\mathrm{Dom}_*(g)$ such that $\mathrm{Dom}(g) = \mathrm{Dom}_*(g) \times \mathbb{N}$ (3) $|g(x,\ell)| = \ell$ for all $(x,\ell) \in \mathrm{Dom}(g)$, and (4) $g(x,\ell) \preceq g(x,\ell')$ whenever $\ell \leq \ell'$. We let $\mathrm{XOL}(\mathcal{D})$ denote the set of all xol functions $g\colon\ \mathcal{D} \to \{0,1\}^*$.

4 Read-only indifferentiability of translating functors

We define read-only indifferentiability (rd-indff) of functors. Then we define a class of functors called translating, and give general results about their rd-indiff security. Later we will apply this to analyze the security of cloning functors, but the treatment in this section is broader and, looking ahead to possible future applications, more general than we need for ours.

4.1 Functors and read-only indifferentiability

A random oracle, formally, is a function drawn at random from a certain space of functions. A construction (functor) is a mapping from one such space to another. We start with definitions for these.

FUNCTION SPACES AND FUNCTORS. A function space FS is simply a set of functions, with the requirement that all functions in the set have the same domain $\mathsf{Dom}(\mathsf{FS})$ and the same range $\mathsf{Rng}(\mathsf{FS})$. Examples are $\mathrm{SOL}(\mathcal{D}, \ell)$ and $\mathrm{XOL}(\mathcal{D})$. Now $f \leftarrow\!\!\$\, \mathsf{FS}$ means we pick a function uniformly at random from the set FS.

Sometimes (but not always) we want an extra condition called input independence. It asks that the values of f on different inputs are identically and independently distributed when $f \leftarrow\!\!\$\, \mathsf{FS}$. More formally, let $\mathcal{D}$ be a set and let Out be a function that associates to any $W \in \mathcal{D}$ a set $\mathrm{Out}(W)$. Let $\mathrm{Out}(\mathcal{D})$ be the union of the sets $\mathrm{Out}(W)$ as W ranges over $\mathcal{D}$. Let $\mathrm{FUNC}(\mathcal{D}, \mathrm{Out})$ be the set of all functions $f\colon\ \mathcal{D} \to \mathrm{Out}(\mathcal{D})$ such that $f(W) \in \mathrm{Out}(W)$ for all $W \in \mathcal{D}$. We say that FS provides input independence if there exists such a Out such that $\mathsf{FS} = \mathrm{FUNC}(\mathsf{Dom}(\mathsf{FS}), \mathrm{Out})$. Put another way, there is a bijection between FS and the set S that is the cross product of the sets $\mathrm{Out}(W)$ as W ranges over $\mathsf{Dom}(\mathsf{FS})$. (Members of S are $|\mathsf{Dom}(\mathsf{FS})|$-vectors.) As an example the function space $\mathrm{SOL}(\mathcal{D}, \ell)$ satisfies input independence, but $\mathrm{XOL}(\mathcal{D})$ does *not* satisfy input independence.

Let SS be a function space that we call the starting space. Let ES be another function space that we call the ending space. We imagine that we are given a function $s \in \mathsf{SS}$ and want to construct a function $e \in \mathsf{ES}$. We refer to the object doing this as a functor. Formally a *functor* is a deterministic algorithm $\mathbf{F}$ that, given as oracle a function $s \in \mathsf{SS}$, returns a function $\mathbf{F}[s] \in \mathsf{ES}$. We write $\mathbf{F}\colon\ \mathsf{SS} \to \mathsf{ES}$ to emphasize the starting and ending spaces of functor $\mathbf{F}$.

RD-INDIFF. We want the ending function to "emulate" a random function from ES. Indifferentiability is a way of defining what this means. The original definition of MRH [29] has been followed by many variants [17, 39, 20, 33]. Here we give ours, called read-only indifferentiability, which implies composition not just for single-stage games, but even for multi-stage ones [39, 20, 33].

Let ES and SS be function spaces, and let $\mathbf{F}\colon\ \mathsf{SS} \to \mathsf{ES}$ be a functor. Our variant of indifferentiability mandates a particular, strong simulator, which can read, but not write, its (game-maintained) state, so that this state is a static quantity. Formally a *read-only simulator* S for $\mathbf{F}$ specifies a *setup algorithm* $\mathsf{S}.\mathrm{Setup}$ which outputs the state, and a deterministic *evaluation algorithm* $\mathsf{S}.\mathrm{Ev}$

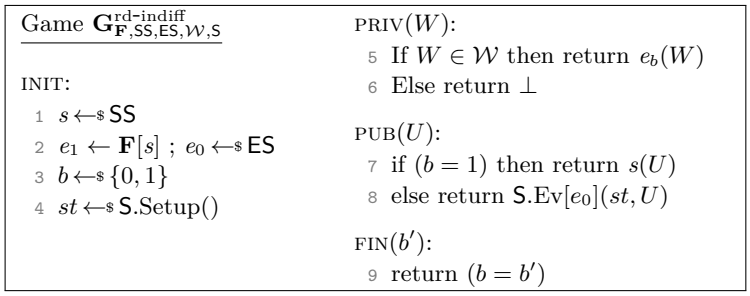

Fig. 2. Game defining read-only indifferentiability.

that, given as oracle a function $e \in \mathsf{ES}$, and given a string $st \in \text{OUT}(\mathsf{S}.\text{Setup})$ (the read-only state), defines a function $\mathsf{S}.\text{Ev}[e](st, \cdot)$: $\text{Dom}(\mathsf{SS}) \to \text{Rng}(\mathsf{SS})$.

The intent is that $\mathsf{S}.\text{Ev}[e](st, \cdot)$ play the role of a starting function $s \in \mathsf{SS}$ satisfying $\mathbf{F}[s] = e$. To formalize this, consider the read-only indifferentiability game $\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathcal{W},\mathsf{S}}$ of Figure 2, where $\mathcal{W} \subseteq \text{Dom}(\mathsf{ES})$ is called the working domain. The adversary $\mathcal{A}$ playing this game is called a distinguisher. Its advantage is defined as

$$\mathbf{Adv}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathcal{W},\mathsf{S}}(\mathcal{A}) = 2 \cdot \Pr\left[\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathcal{W},\mathsf{S}}(\mathcal{A})\right] - 1.$$

To explain, in the game, b is a challenge bit that the distinguisher is trying to determine. Function e_b is a random member of the ending space ES if $b = 0$ and is $\mathbf{F}[s](\cdot)$ if $b = 1$. The query W to oracle PRIV is required to be in $\text{Dom}(\mathsf{ES})$. The oracle returns the value of e_b on W, but only if W is in the working domain, otherwise returning $\perp$. The query U to oracle PUB is required to be in $\text{Dom}(\mathsf{SS})$. The oracle returns the value of s on U in the $b = 1$ case, but when $b = 0$, the simulator evaluation algorithm $\mathsf{S}.\text{Ev}$ must answer the query with access to an oracle for e_0. The distinguisher ends by calling FIN with its guess $b' \in \{0,1\}$ of b and the game returns true if $b' = b$ (the distinguisher's guess is correct) and false otherwise.

The working domain $\mathcal{W} \subseteq \text{Dom}(\mathsf{ES})$, a parameter of the definition, is included as a way to allow the notion of read-only indifferentiability to provide results for oracle cloning methods like length differentiation whose security depends on domain restrictions.

The $\mathsf{S}.\text{Ev}$ algorithm is given direct access to e_0, rather than access to PRIV as in other definitions, to bypass the working domain restriction, meaning it may query e_0 at points in $\text{Dom}(\mathsf{ES})$ that are outside the working domain.

All invocations of $\mathsf{S}.\text{Ev}[e_0]$ are given the same (static, game-maintained) state st as input, but $\mathsf{S}.\text{Ev}[e_0]$ cannot modify this state, which is why it is called read-only. Note INIT does not return st, meaning the state is not given to the distinguisher.

Discussion. To compare rd-indiff to other indiff notions, we set $\mathcal{W} = \text{Dom}(\mathsf{ES})$, because prior notions do not include working domains. Now, rd-indiff differs from prior indiff notions because it requires that the simulator state be just

the immutable string chosen at the start of the game. In this regard, rd-indiff falls somewhere between the original MRH-indiff [29] and reset indiff [39] in the sense that our simulator is more restricted than in the first and less than in the second. A construction (functor) that is reset-indiff is thus rd-indiff, but not necessarily vice-versa, and a construct that is rd-indiff is MRH-indiff, but not necessarily vice-versa. Put another way, the class of rd-indff functors is larger than the class of reset-indiff ones, but smaller than the class of MRH-indiff ones. Now, RSS's proof [39] that reset-indiff implies security for multi-stage games extends to rd-indiff, so we get this for a potentially larger class of functors. This larger class includes some of the cloning functors we have described, which are not necessarily reset-indiff.

4.2 Translating functors

Translating functors. We focus on a class of functors that we call translating. This class includes natural and existing oracle cloning methods, in particular all the effective methods used by NIST KEMs, and we will be able to prove general results for translating functors that can be applied to the cloning methods.

A translating functor $\mathbf{T}\colon \mathsf{SS} \to \mathsf{ES}$ is a functor that, with oracle access to s and on input $W \in \mathrm{Dom}(\mathsf{ES})$, non-adaptively calls s on a fixed number of inputs, and computes its output $\mathbf{T}[s](W)$ from the responses and W. Its operation can be split into three phases which do not share state: (1) a pre-processing phase which chooses the inputs to s based on W alone (2) the calls to s to obtain responses (3) a post-processing phase which uses W and the responses collected in phase 2 to compute the final output value $\mathbf{T}[s](W)$.

Proceeding to the definitions, let SS, ES be function spaces. A $(\mathsf{SS}, \mathsf{ES})$-*query translator* is a function (deterministic algorithm) $\mathsf{QT}\colon \mathrm{Dom}(\mathsf{ES}) \to \mathrm{Dom}(\mathsf{SS})^*$, meaning it takes a point W in the domain of the ending space and returns a vector of points in the domain of the starting space. This models the pre-processing. A $(\mathsf{SS}, \mathsf{ES})$-*answer translator* is a function (deterministic algorithm) $\mathsf{AT}\colon \mathrm{Dom}(\mathsf{ES}) \times \mathrm{Rng}(\mathsf{SS})^* \to \mathrm{Rng}(\mathsf{ES})$, meaning it takes the original W, and a vector of points in the range of the starting space, to return a point in the range of the ending space. This models the post-processing. To the pair $(\mathsf{QT}, \mathsf{AT})$, we associate the functor $\mathbf{TF}_{\mathsf{QT},\mathsf{AT}}\colon \mathsf{SS} \to \mathsf{ES}$, defined as follows:

Algorithm $\mathbf{TF}_{\mathsf{QT},\mathsf{AT}}[s](W)$ // Input $W \in \mathrm{Dom}(\mathsf{ES})$ and oracle $s \in \mathsf{SS}$
$\boldsymbol{U} \leftarrow \mathsf{QT}(W)$
For $j = 1, \ldots, |\boldsymbol{U}|$ do $\boldsymbol{V}[j] \leftarrow s(\boldsymbol{U}[j])$ // $\boldsymbol{U}[j] \in \mathsf{Dom}(\mathsf{SS})$
$Y \leftarrow \mathsf{AT}(W, \boldsymbol{V})$; Return Y

The above-mentioned calls of phase (2) are done in the second line of the code above, so that this implements a translating functor as we described. Formally we say that a functor $\mathbf{F}\colon \mathsf{SS} \to \mathsf{ES}$ is *translating* if there exists a $(\mathsf{SS}, \mathsf{ES})$-query translator QT and a $(\mathsf{SS}, \mathsf{ES})$-answer translator AT such that $\mathbf{F} = \mathbf{TF}_{\mathsf{QT},\mathsf{AT}}$.

Inverses. So far, query and answer translators may have just seemed an unduly complex way to say that a translating oracle construction is one that makes non-

adaptive oracle queries. The purpose of making the query and answer translators explicit is to define *invertibility*, which determines rd-indiff security.

Let SS and ES be function spaces. Let QTI be a function (deterministic algorithm) that takes an input $U \in \mathrm{Dom}(\mathsf{SS})$ and returns a vector $\boldsymbol{W}$ over $\mathrm{Dom}(\mathsf{ES})$. We allow QTI to return the empty vector $()$, which is taken as an indication of failure to invert. Define the *support* of QTI, denoted $\mathbf{sup}(\mathsf{QTI})$, to be the set of all $U \in \mathrm{Dom}(\mathsf{SS})$ such that $\mathsf{QTI}(U) \neq ()$. Say that QTI has *full support* if $\mathbf{sup}(\mathsf{QTI}) = \mathrm{Dom}(\mathsf{SS})$, meaning there is no $U \in \mathrm{Dom}(\mathsf{SS})$ such that $\mathsf{QTI}(U) = ()$. Let ATI be a function (deterministic algorithm) that takes $U \in \mathrm{Dom}(\mathsf{SS})$ and a vector $\boldsymbol{Y}$ over $\mathrm{Rng}(\mathsf{ES})$ to return an output in $\mathrm{Rng}(\mathsf{SS})$. Given a function $e \in \mathsf{ES}$, we define the function $\mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}\colon\ \mathrm{Dom}(\mathsf{SS}) \to \mathrm{Rng}(\mathsf{SS})$ by

$$\underline{\text{Function } \mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}(U)} \quad /\!\!/\ U \in \mathrm{Dom}(\mathsf{SS})$$
$$\boldsymbol{W} \leftarrow \mathsf{QTI}(U)\ ;\ \boldsymbol{Y} \leftarrow e(\boldsymbol{W})\ ;\ V \leftarrow \mathsf{ATI}(U, \boldsymbol{Y})\ ;\ \text{Return } V$$

Above, e is applied to a vector component-wise, meaning $e(\boldsymbol{W})$ is defined as the vector $(e(\boldsymbol{W}[1]), \ldots, e(\boldsymbol{W}[|\boldsymbol{W}|]))$.

We require that the function $\mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}$ belong to the starting space SS. Now let QT be a $(\mathsf{SS},\mathsf{ES})$-query translator and AT a $(\mathsf{SS},\mathsf{ES})$-answer translator. Let $\mathcal{W} \subseteq \mathrm{Dom}(\mathsf{ES})$ be a working domain. We say that $\mathsf{QTI},\mathsf{ATI}$ *are inverses of* QT,AT *over* $\mathcal{W}$ if two conditions are true. The first is that for all $e \in \mathsf{ES}$ and all $W \in \mathcal{W}$ we have

$$\mathbf{TF}_{\mathsf{QT},\mathsf{AT}}[\mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}](W) = e(W)\ . \tag{2}$$

This equation needs some parsing. Fix a function $e \in \mathsf{ES}$ in the ending space. Then $s = \mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}$ is in SS. Recall that the functor $\mathbf{F} = \mathbf{TF}_{\mathsf{QT},\mathsf{AT}}$ takes a function s in the starting space as an oracle and defines a function $e' = \mathbf{F}[s]$ in the ending space. Equation (2) is asking that e' is identical to the original function e, on the working domain $\mathcal{W}$. The second condition (for invertibility) is that if $U \in \{\mathsf{QT}(W)[i]\ :\ W \in \mathcal{W}\}$ —that is, U is an entry of the vector $\boldsymbol{U}$ returned by QT on some input W— then $\mathsf{QTI}(U) \neq ()$. Note that if QTI has full support then this condition is already true, but otherwise it is an additional requirement.

We say that $(\mathsf{QT},\mathsf{AT})$ is invertible over $\mathcal{W}$ if there exist $\mathsf{QTI},\mathsf{ATI}$ such that $\mathsf{QTI},\mathsf{ATI}$ are inverses of QT,AT over $\mathcal{W}$, and we say that a translating functor $\mathbf{TF}_{\mathsf{QT},\mathsf{AT}}$ is invertible over $\mathcal{W}$ if $(\mathsf{QT},\mathsf{AT})$ is invertible over $\mathcal{W}$.

In the rd-indiff context, function $\mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}$ will be used by the simulator. Roughly, we try to set $\mathsf{S}.\mathrm{Ev}[e](st, U) = \mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}(U)$. But we will only be able to successfully do this for $U \in \mathbf{sup}(\mathsf{QTI})$. The state st is used by $\mathsf{S}.\mathrm{Ev}$ to provide replies when $U \notin \mathbf{sup}(\mathsf{QTI})$.

Equation (2) is a correctness condition. There is also a security metric. Consider the *translation indistinguishability* game $\mathbf{G}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}$ of Figure 3. Define the ti-advantage of adversary $\mathcal{B}$ via

$$\mathbf{Adv}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}(\mathcal{B}) = 2 \cdot \Pr\left[\mathbf{G}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}(\mathcal{B})\right] - 1.$$

In reading the game, recall that $()$ is the empty vector, whose return by QTI represents an inversion error. TI-security is thus asking that if e is randomly chosen from the ending space, then the output of $\mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}$ on an input U is

Game $\mathbf{G}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}$

INIT:
1 $b \xleftarrow{\$} \{0,1\}$; $e \xleftarrow{\$} \mathsf{ES}$
2 $s_1 \xleftarrow{\$} \mathsf{SS}$; $s_0 \leftarrow \mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}$

PUB(U): // $U \in \mathrm{Dom}(\mathsf{SS})$
3 If $\mathsf{QTI}(U) = ()$ then return $\bot$
4 return $s_b(U)$

FIN(b'):
5 return $(b = b')$

Fig. 3. Game defining translation indistinguishability.

Algorithm S.Setup:	Algorithm $\mathsf{S}.\mathrm{Ev}[e](st, U)$:
1 Return ε	1 $\boldsymbol{W} \leftarrow \mathsf{QTI}(U)$; $\boldsymbol{Y} \leftarrow e(\boldsymbol{W})$; $V \leftarrow \mathsf{ATI}(U, \boldsymbol{Y})$ 2 Return V

Algorithm S.Setup:	Algorithm $\mathsf{S}.\mathrm{Ev}[e](st, U)$:
1 $st \xleftarrow{\$} \{0,1\}^{\mathsf{G.kl}}$ 2 Return st	1 $\boldsymbol{W} \leftarrow \mathsf{QTI}(U)$ 2 If $\boldsymbol{W} = ()$ then return $\mathsf{G}_{st}[e](U)$ 3 $\boldsymbol{Y} \leftarrow e(\boldsymbol{W})$; $V \leftarrow \mathsf{ATI}(U, \boldsymbol{Y})$ 4 Return V

Fig. 4. Simulators for Theorem 1 (top) and Theorem 2 (bottom).

distributed like the output on U of a random function in the starting space, *but only as long as* $\mathsf{QTI}(U)$ *was non-empty*. We will see that the latter restriction creates some challenges in simulation whose resolution exploits using read-only state. We say that $(\mathsf{QTI}, \mathsf{ATI})$ provides perfect translation indistinguishability if $\mathbf{Adv}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}(\mathcal{B}) = 0$ for all $\mathcal{B}$, regardless of the running time of $\mathcal{B}$.

Additionally we of course ask that the functions $\mathsf{QT}, \mathsf{AT}, \mathsf{QTI}, \mathsf{ATI}$ all be efficiently computable. In an asymptotic setting, this means they are polynomial time. In our concrete setting, they show up in the running-time of the simulator or constructed adversaries. (The latter, as per our conventions, being the time for the execution of the adversary with the overlying game.)

4.3 Rd-indiff of translating functors

We now move on to showing that invertibility of a pair $(\mathsf{QT}, \mathsf{AT})$ implies rd-indifferentiability of the translating functor $\mathbf{TF}_{\mathsf{QT},\mathsf{AT}}$. We start with the case that QTI has full support.

Theorem 1. *Let* SS *and* ES *be function spaces. Let* $\mathcal{W}$ *be a subset of* $\mathrm{Dom}(\mathsf{ES})$. *Let* QT, AT *be* $(\mathsf{SS}, \mathsf{ES})$ *query and answer translators, respectively. Let* $\mathsf{QTI}, \mathsf{ATI}$ *be inverses of* QT, AT *over* $\mathcal{W}$. *Assume* QTI *has full support. Define read-only*

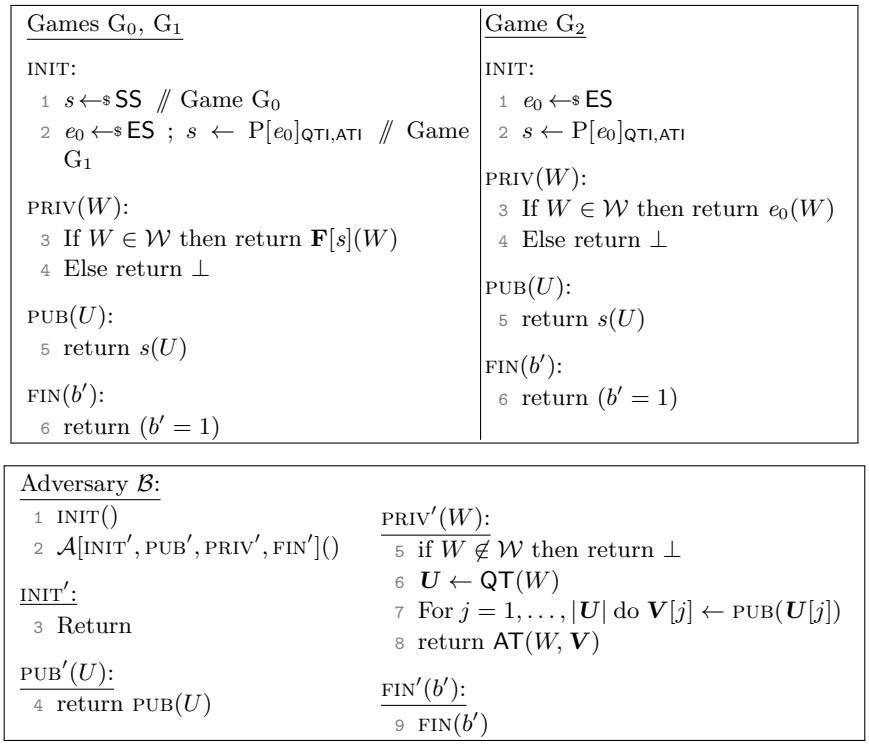

Fig. 5. Top: Games for proof of Theorem 1. Bottom: Adversary for proof of Theorem 1.

simulator S *as per the top panel of Figure 4. Let* $\mathbf{F} = \mathbf{TF}_{\mathsf{QT},\mathsf{AT}}$. *Let* $\mathcal{A}$ *be any distinguisher. Then we construct a ti-adversary* $\mathcal{B}$ *such that*

$$\mathbf{Adv}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathcal{W},\mathsf{S}}(\mathcal{A}) \le \mathbf{Adv}^{\text{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}(\mathcal{B}) \ .$$

Let ℓ *be the maximum output length of* QT. *If* $\mathcal{A}$ *makes* $q_{\text{PRIV}}, q_{\text{PUB}}$ *queries to its* PRIV, PUB *oracles, respectively, then* $\mathcal{B}$ *makes* $\ell \cdot q_{\text{PRIV}} + q_{\text{PUB}}$ *queries to its* PUB *oracle. The running time of* $\mathcal{B}$ *is about that of* $\mathcal{A}$.

Proof (Theorem 1). Consider the games of Figure 5. In the left panel, line 1 is included only in G_0 and line 2 only in G_1, and this is the only way the games differ. Game G_0 is the real game, meaning the case $b = 1$ in game $\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathcal{W},\mathsf{S}}$. In game G_2, oracle PRIV is switched to a random function e_0. From the description of the simulator in Figure 4 we see that

$$\mathsf{S}.\mathrm{Ev}[e_0](\varepsilon, U) = \mathrm{P}[e_0]_{\mathsf{QTI},\mathsf{ATI}}(U)$$

for all $U \in \mathrm{Dom}(\mathsf{SS})$ and all $e_0 \in \mathsf{ES}$, so that oracle PUB in game G_2 is responding according to the simulator based on e_0. So game G_2 is the case $b = 0$ in game

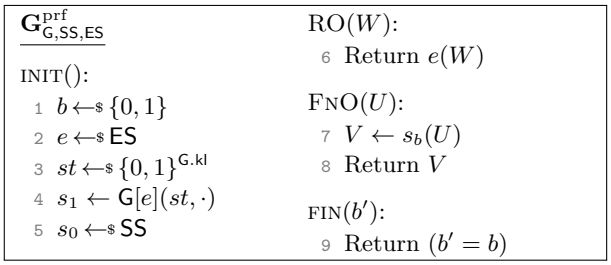

Fig. 6. Game to define PRF security of $(\mathsf{SS},\mathsf{ES})$-oracle aided PRF G.

$\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathcal{W},\mathsf{S}}$. Thus

$$\begin{aligned}\mathbf{Adv}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathcal{W},\mathsf{S}}(\mathcal{A}) &= \Pr[\mathrm{G}_0(\mathcal{A})] - \Pr[\mathrm{G}_2(\mathcal{A})] \\ &= (\Pr[\mathrm{G}_0(\mathcal{A})] - \Pr[\mathrm{G}_1(\mathcal{A})]) + (\Pr[\mathrm{G}_1(\mathcal{A})] - \Pr[\mathrm{G}_2(\mathcal{A})]) \ .\end{aligned}$$

We define translation-indistinguishability adversary $\mathcal{B}$ in Figure 5 so that

$$\Pr[\mathrm{G}_0(\mathcal{A})] - \Pr[\mathrm{G}_1(\mathcal{A})] \le \mathbf{Adv}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}(\mathcal{B}) \ .$$

Adversary $\mathcal{B}$ is playing game $\mathbf{G}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}$. Using its PUB oracle, it presents the interface of G_0 and G_1 to $\mathcal{A}$. In order to simulate the PRIV oracle, $\mathcal{B}$ runs $\mathbf{TF}_{\mathsf{QT},\mathsf{AT}}[\mathrm{PUB}]$. This is consistent with G_0 and G_1. If $b = 1$ in $\mathbf{G}^{\mathrm{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}$, then $\mathcal{B}$ perfectly simulates G_0 for $\mathcal{A}$. If $b = 1$, then $\mathcal{B}$ correctly simulates G_1 for $\mathcal{A}$. To complete the proof we claim that

$$\Pr[\mathrm{G}_1(\mathcal{A})] = \Pr[\mathrm{G}_2(\mathcal{A})] \ .$$

This is true by the correctness condition. The latter says that if $s \leftarrow \mathrm{P}[e_0]_{\mathsf{QTI},\mathsf{ATI}}$ then $\mathbf{F}[s]$ is just e_0 itself. So e_1 in game G_1 is the same as e_0 in game G_2, making their PRIV oracles identical. And their PUB oracles are identical by definition. □

The simulator in Theorem 1 is stateless, so when $\mathcal{W}$ is chosen to be $\mathrm{Dom}(\mathsf{ES})$ the theorem is establishing reset indifferentiability [39] of $\mathbf{F}$.

For translating functors where QTI does not have full support, we need an auxiliary primitive that we call a $(\mathsf{SS},\mathsf{ES})$-oracle aided PRF. Given an oracle for a function $e \in \mathsf{ES}$, an $(\mathsf{SS},\mathsf{ES})$-oracle aided PRF G defines a function $\mathsf{G}[e]\colon\ \{0,1\}^{\mathsf{G.kl}} \times \mathrm{Dom}(\mathsf{SS}) \to \mathrm{Rng}(\mathsf{SS})$. The first input is a key. For $\mathcal{C}$ an adversary, let $\mathbf{Adv}^{\mathrm{prf}}_{\mathsf{G},\mathsf{SS},\mathsf{ES}}(\mathcal{C}) = 2\Pr[\mathbf{G}^{\mathrm{prf}}_{\mathsf{G},\mathsf{SS},\mathsf{ES}}(\mathcal{C})] - 1$, where the game is in Figure 6. The simulator uses its read-only state to store a key st for G, then using $\mathsf{G}(st,\cdot)$ to answer queries outside the support $\mathbf{sup}(\mathsf{QTI})$.

We introduce this primitive because it allows multiple instantiations. The simplest is that it is a PRF, which happens when it does not use its oracle. In that case the simulator is using a computational primitive (a PRF) in the indifferentiability context, which seems novel. Another instantiation prefixes st to the input and then invokes e to return the output. This works for certain choices of ES, but not always. Note G is used only by the simulator and plays no role in the functor.

The proof of the following is in [10].

Theorem 2. *Let* SS *and* ES *be function spaces, and assume they provide input independence. Let* $\mathcal{W}$ *be a subset of* $\mathrm{Dom}(\mathsf{ES})$*. Let* QT, AT *be* $(\mathsf{SS}, \mathsf{ES})$ *query and answer translators, respectively. Let* $\mathsf{QTI}, \mathsf{ATI}$ *be inverses of* QT, AT *over* $\mathcal{W}$*. Define read-only simulator* S *as per the bottom panel of Figure 4. Let* $\mathbf{F} = \mathbf{TF}_{\mathsf{QT},\mathsf{AT}}$*. Let* $\mathcal{A}$ *be any distinguisher. Then we construct a ti-adversary* $\mathcal{B}$ *and a prf-adversary* $\mathcal{C}$ *such that*

$$\mathbf{Adv}^{\text{rd-indiff}}_{\mathbf{F},\mathsf{SS},\mathsf{ES},\mathcal{W},\mathsf{S}}(\mathcal{A}) \leq \mathbf{Adv}^{\text{ti}}_{\mathsf{SS},\mathsf{ES},\mathsf{QTI},\mathsf{ATI}}(\mathcal{B}) + \mathbf{Adv}^{\text{prf}}_{\mathsf{G},\mathsf{SS}}(\mathcal{C}) \ .$$

Let ℓ *be the maximum output length of* QT *and* ℓ' *the maximum output length of* QTI*. If* $\mathcal{A}$ *makes* $q_{\text{PRIV}}, q_{\text{PUB}}$ *queries to its* PRIV, PUB *oracles, respectively, then* $\mathcal{B}$ *makes* $\ell \cdot q_{\text{PRIV}} + q_{\text{PUB}}$ *queries to its* PUB *oracle and* $\mathcal{C}$ *makes at most* $\ell \cdot \ell' \cdot q_{\text{PRIV}} + q_{\text{PUB}}$ *queries to its* RO *oracle and at most* $q_{\text{PUB}} + \ell \cdot q_{\text{PRIV}}$ *queries to its* FnO *oracle. The running times of* $\mathcal{B}, \mathcal{C}$ *are about that of* $\mathcal{A}$*.*

5 Analysis of cloning functors

Section 4 defined the rd-indiff metric of security for functors and give a framework to prove rd-indiff of translating functors. We now apply this to derive security results about particular, practical cloning functors.

Arity-n function spaces. The cloning functors apply to function spaces where a function specifies sub-functions, corresponding to the different random oracles we are trying to build. Formally, a function space FS is said to have arity n if its members are two-argument functions f whose first argument is an integer $i \in [1..n]$. For $i \in [1..n]$ we let $f_i = f(i, \cdot)$ and $\mathsf{FS}_i = \{f_i : f \in \mathsf{FS}\}$, and refer to the latter as the i-th subspace of FS. We let $\mathsf{Dom}_i(\mathsf{FS})$ be the set of all X such that $(i, X) \in \mathrm{Dom}(\mathsf{FS})$.

We say that FS has sol subspaces if FS_i is a set of sol functions with domain $\mathsf{Dom}_i(\mathsf{FS})$, for all $i \in [1..n]$. More precisely, there must be integers $\mathsf{OL}_1(\mathsf{FS}), \ldots, \mathsf{OL}_n(\mathsf{FS})$ such that $\mathsf{FS}_i = \mathrm{SOL}(\mathsf{Dom}_i(\mathsf{FS}), \mathsf{OL}_i(\mathsf{FS}))$ for all $i \in [1..n]$. In this case, we let $\mathsf{Rng}_i(\mathsf{FS}) = \{0,1\}^{\mathsf{OL}_i(\mathsf{FS})}$. This is the most common case for practical uses of ROs.

To explain, access to n random oracles is modeled as access to a two-argument function f drawn at random from FS, written $f \leftarrow\!\!\$\ \mathsf{FS}$. If FS has sol subspaces, then for each i, the function f_i is a sol function, with a certain domain and output length depending only on i. All such functions are included. This ensures input independence as we defined it earlier. Thus if $f \leftarrow\!\!\$\ \mathsf{FS}$, then for each i and any distinct inputs to f_i, the outputs are independently distributed. Also functions $f_1, \ldots, f_n$ are independently distributed when $f \leftarrow\!\!\$\ \mathsf{FS}$. Put another way, we can identify FS with $\mathsf{FS}_1 \times \cdots \times \mathsf{FS}_n$.

Domain-separating functors. We can now formalize the domain separation method by seeing it as defining a certain type of (translating) functor.

Let the ending space ES be an arity n function space. Let $\mathbf{F}\colon\ \mathsf{SS} \to \mathsf{ES}$ be a translating functor and QT, AT be its query and answer translations, respectively.

Assume QT returns a vector of length 1 and that $\mathsf{AT}((i, X), \boldsymbol{V})$ simply returns $\boldsymbol{V}[1]$. We say that $\mathbf{F}$ is *domain separating* if the following is true: $\mathsf{QT}(i_1, X_1) \neq \mathsf{QT}(i_2, X_2)$ for any $(i_1, X_1), (i_2, X_2) \in \mathrm{Dom}(\mathsf{ES})$ that satisfy $i_1 \neq i_2$.

To explain, recall that the ending function is obtained as $e \leftarrow \mathbf{F}[s]$, and defines e_i for $i \in [1..n]$. Function e_i takes input X, lets $(u) \leftarrow \mathsf{QT}(i, X)$ and returns $s(u)$. The domain separation requirement is that if $(u_i) \leftarrow \mathsf{QT}(i, X_i)$ and $(u_j) \leftarrow \mathsf{QT}(j, X_j)$, then $i \neq j$ implies $u_i \neq u_j$, regardless of X_i, X_j. Thus if $i \neq j$ then the inputs to which s is applied are always different. The domain of s has been "separated" into disjoint subsets, one for each i.

Practical cloning functors. We show that many popular methods for oracle cloning in practice, including ones used in NIST KEM submissions, can be cast as translating functors.

In the following, the starting space $\mathsf{SS} = \mathrm{SOL}(\{0,1\}^*, \mathsf{OL}(\mathsf{SS}))$ is assumed to be a sol function space with domain $\{0,1\}^*$ and an output length denoted $\mathsf{OL}(\mathsf{SS})$. The ending space ES is an arity n function spaces that has sol subspaces.

Prefixing. Here we formalize the canonical method of domain separation. Prefixing is used in the following NIST PQC submissions: **ClassicMcEliece**, **FrodoKEM**, LIMA, **NTRU Prime**, SIKE, QC-MDPC, **ThreeBears**.

Let $\mathbf{p}$ be a vector of strings. We require that it be *prefix-free*, by which we mean that $i \neq j$ implies that $\mathbf{p}[i]$ is not a prefix of $\mathbf{p}[j]$. Entries of this vector will be used as prefixes to enforce domain separation. One example is that the entries of $\mathbf{p}$ are distinct strings all of the same length. Another is that a $\mathbf{p}[i] = \mathrm{E}(i)$ for some prefix-free code E like a Huffman code.

Assume $\mathsf{OL}_i(\mathsf{ES}) = \mathsf{OL}(\mathsf{SS})$ for all $i \in [1..n]$, meaning all ending functions have the same output length as the starting function. The functor $\mathbf{F}_{\mathrm{pf}(\mathbf{p})}\colon \mathsf{SS} \to \mathsf{ES}$ corresponding to $\mathbf{p}$ is defined by $\mathbf{F}_{\mathrm{pf}(\mathbf{p})}[s](i, X) = s(\mathbf{p}[i] \| X)$. To explain, recall that the ending function is obtained as $e \leftarrow \mathbf{F}_{\mathrm{pf}(\mathbf{p})}[s]$, and defines e_i for $i \in [1..n]$. Function e_i takes input X, prefixes $\mathbf{p}[i]$ to X to get a string X', applies the starting function s to X' to get Y, and returns Y as the value of $e_i(X)$.

We claim that $\mathbf{F}_{\mathrm{pf}(\mathbf{p})}$ is a translating functor that is also a domain-separating functor as per the definitions above. To see this, define query translator $\mathsf{QT}_{\mathrm{pf}(\mathbf{p})}$ by $\mathsf{QT}_{\mathrm{pf}(\mathbf{p})}(i, X) = (\mathbf{p}[i] \| X)$, the 1-vector whose sole entry is $\mathbf{p}[i] \| X$. The answer translator $\mathsf{AT}_{\mathrm{pf}(\mathbf{p})}$, on input (i, X), $\boldsymbol{V}$, returns $\boldsymbol{V}[1]$, meaning it ignores i, X and returns the sole entry in its 1-vector $\boldsymbol{V}$.

We proceed to the inverses, which are defined as follows:

Algorithm $\mathsf{QTI}_{\mathrm{pf}(\mathbf{p})}(U)$	Algorithm $\mathsf{ATI}_{\mathrm{pf}(\mathbf{p})}(U, \boldsymbol{Y})$
$\boldsymbol{W} \leftarrow ()$	If $\boldsymbol{Y} \neq ()$ then $V \leftarrow \boldsymbol{Y}[1]$
For $i = 1, \ldots, n$ do	Else $V \leftarrow 0^{\mathsf{OL}(\mathsf{SS})}$
If $\mathbf{p}[i] \preceq U$ then $\mathbf{p}[i] \| X \leftarrow U$; $\boldsymbol{W}[1] \leftarrow (i, X)$	Return V
Return $\boldsymbol{W}$	

The working domain is the full one: $\mathcal{W} = \mathrm{Dom}(\mathsf{ES})$. We now verify Equation (2). Let $\mathsf{QT}, \mathsf{QTI}, \mathsf{AT}, \mathsf{ATI}$ be $\mathsf{QT}_{\mathrm{pf}(\mathbf{p})}, \mathsf{QTI}_{\mathrm{pf}(\mathbf{p})}, \mathsf{AT}_{\mathrm{pf}(\mathbf{p})}, \mathsf{ATI}_{\mathrm{pf}(\mathbf{p})}$, respectively. Then

for all $W = (i, X) \in \mathrm{Dom}(\mathsf{ES})$, we have:

$$\begin{aligned}\mathbf{TF}_{\mathsf{QT},\mathsf{AT}}[\mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}](W) &= \mathrm{P}[e]_{\mathsf{QTI},\mathsf{ATI}}(\mathbf{p}[i]\|X) \\ &= \mathsf{ATI}(\mathbf{p}[i]\|X, (e(i,X))) \\ &= e(i,X)\ .\end{aligned}$$

We observe that $(\mathsf{QTI}_{\mathrm{pf}(\mathbf{p})}, \mathsf{ATI}_{\mathrm{pf}(\mathbf{p})})$ provides perfect translation indistinguishability. Since $\mathsf{QTI}_{\mathrm{pf}(\mathbf{p})}$ does not have full support, we can't use Theorem 1, but we can conclude rd-indiff via Theorem 2.

Identity. Many NIST PQC submissions simply let $e_i(X) = s(X)$, meaning the ending functions are identical to the starting one. This is captured by the identity functor $\mathbf{F}_{\mathrm{id}}$: $\mathsf{SS} \to \mathsf{ES}$, defined by $\mathbf{F}_{\mathrm{id}}[s](i, X) = s(X)$. This again assumes $\mathsf{OL}_i(\mathsf{ES}) = \mathsf{OL}(\mathsf{SS})$ for all $i \in [1..n]$, meaning all ending functions have the same output length as the starting function. This functor is translating, via $\mathsf{QT}_{\mathrm{id}}(i, X) = X$ and $\mathsf{AT}_{\mathrm{id}}((i, X), \boldsymbol{V}) = \boldsymbol{V}[1]$. It is however *not*, at least in general, domain separating.

Clearly, this functor is not, in general, rd-indiff. To make secure use of it nonetheless, applications can restrict the inputs to the ending functions to enforce a virtual domain separation, meaning, for $i \neq j$, the schemes never query e_i and e_j on the same input. One way to do this is length differentiation. Here, for $i \in [1..n]$, the inputs to which e_i is applied all have the same length l_i, and $l_1, \ldots, l_n$ are distinct. Length differentiation is used in the following NIST PQC submissions: **BIKE**,EMBLEM, **HQC**, **RQC**, **LAC**, LOCKER, **NTS-KEM**, **SABER**, Round2, **Round5**,Titanium. There are, of course, many other similar ways to enforce the virtual domain separation.

There are two ways one might capture this with regard to security. One is to restrict the domain $\mathrm{Dom}(\mathsf{ES})$ of the ending space. For example, for length differentiation, we would require that there exist distinct $l_1, \ldots, l_n$ such that for all $(i, X) \in \mathrm{Dom}(\mathsf{ES})$ we have $|X| = l_i$. For such an ending space, the identity functor would provide security. The approach we take is different. We don't restrict the domain of the ending space, but instead define security with respect to a subdomain, which we called the working space, where the restriction is captured. This, we believe, is better suited for practice, for a few reasons. One is that a single implementation of the ending functions can be used securely in different applications that each have their own working domain. Another is that implementations of the ending functions do not appear to enforce any restrictions, leaving it up to applications to figure out how to securely use the functions. In this context, highlighting the working domain may help application designers think about what is the working domain in their application and make this explicit, which can reduce error.

But we warn that the identity functor approach is more prone to misuse and in the end more dangerous and brittle than some others.

As per the above, inverses can only be given for certain working domains. Let us say that $\mathcal{W} \subseteq \mathrm{Dom}(\mathsf{ES})$ separates domains if for all $(i_1, X_1), (i_2, X_2) \in \mathcal{W}$ satisfying $i_1 \neq i_2$, we have $X_1 \neq X_2$. Put another way, for any $(i, X) \in \mathcal{W}$

there is at most one j such that $X \in \mathsf{Dom}_j(\mathsf{ES})$. We assume an efficient inverter for $\mathcal{W}$. This is a deterministic algorithm $\mathrm{In}_{\mathcal{W}}$ that on input $X \in \{0,1\}^*$ returns the unique i such that $(i, X) \in \mathcal{W}$ if such an i exists, and otherwise returns $\perp$. (The uniqueness is by the assumption that $\mathcal{W}$ separates domains.)

As an example, for length differentiation, we pick some *distinct* integers $l_1, \ldots, l_n$ such that $\{0,1\}^{l_i} \subseteq \mathsf{Dom}_i(\mathsf{ES})$ for all $i \in [1..n]$. We then let $\mathcal{W} = \{(i, X) \in \mathrm{Dom}(\mathsf{ES}) : |X| = l_i\}$. This separates domains. Now we can define $\mathrm{In}_{\mathcal{W}}(X)$ to return the unique i such that $|X| = l_i$ if $|X| \in \{l_1, \ldots, l_n\}$, otherwise returning $\perp$.

The inverses are then defined using $\mathrm{In}_{\mathcal{W}}$, as follows, where $U \in \mathrm{Dom}(\mathsf{SS}) = \{0,1\}^*$:

Algorithm $\mathsf{QTI}_{\mathrm{id}}(U)$	Algorithm $\mathsf{ATI}_{\mathrm{id}}(U, \boldsymbol{Y})$
$\boldsymbol{W} \leftarrow () \; ; \; i \leftarrow \mathrm{In}_{\mathcal{W}}(U)$	If $\boldsymbol{Y} \neq ()$ then $V \leftarrow \boldsymbol{Y}[1]$
If $i \neq \perp$ then $\boldsymbol{W}[1] \leftarrow (i, U)$	Else $V \leftarrow 0^{\mathsf{OL}(\mathsf{SS})}$
Return $\boldsymbol{W}$	Return V

The correctness condition of Equation (2) over $\mathcal{W}$ is met, and since $\mathrm{In}_{\mathcal{W}}(X)$ never returns $\perp$ for $X \in \mathcal{W}$, the second condition of invertibility is also met. $(\mathsf{QTI}_{\mathrm{id}}, \mathsf{ATI}_{\mathrm{id}})$ provides perfect translation indistinguishability. Since $\mathsf{QTI}_{\mathrm{id}}$ does not have full support, we can't use Theorem 1, but we can conclude rd-indiff via Theorem 2.

Output-splitting. We formalize another method that we call output splitting. It is used in the following NIST PQC submissions: **FrodoKEM**, NTRU-HRSS-KEM, Odd Manhattan,QC-MDPC, Round2, **Round5**.

Let $\ell_i = \mathsf{OL}_1(\mathsf{ES}) + \cdots + \mathsf{OL}_i(\mathsf{ES})$ for $i \in [1..n]$. Let $\ell = \mathsf{OL}(\mathsf{SS})$ be the output length of the sol functions $s \in \mathsf{SS}$, and assume $\ell = \ell_n$. The output-splitting functor $\mathbf{F}_{\mathrm{spl}}$: $\mathsf{SS} \to \mathsf{ES}$ is defined by $\mathbf{F}_{\mathrm{spl}}[s](i, X) = s(X)[\ell_{i-1}+1..\ell_i]$. That is, if $e \leftarrow \mathbf{F}_{\mathrm{spl}}[s]$, then $e_i(X)$ lets $Z \leftarrow s(X)$ and then returns bits $\ell_{i-1}+1$ through ℓ_i of Z. This functor is translating, via $\mathsf{QT}_{\mathrm{spl}}(i, X) = X$ and $\mathsf{AT}_{\mathrm{spl}}((i, X), \boldsymbol{V}) = \boldsymbol{V}[1][\ell_{i-1}+1..\ell_i]$. It is however *not* domain separating.

The inverses are defined as follows, where $U \in \mathrm{Dom}(\mathsf{SS}) = \{0,1\}^*$:

Algorithm $\mathsf{QTI}_{\mathrm{spl}}(U)$	Algorithm $\mathsf{ATI}_{\mathrm{spl}}(U, \boldsymbol{Y})$
For $i = 1, \ldots, n$ do $\boldsymbol{W}[i] \leftarrow (i, U)$	$V \leftarrow \boldsymbol{Y}[1] \| \cdots \| \boldsymbol{Y}[n]$
Return $\boldsymbol{W}$	Return V

The correctness condition of Equation (2) over $\mathcal{W} = \mathsf{ES}$ is met, and $(\mathsf{QTI}_{\mathrm{spl}}, \mathsf{ATI}_{\mathrm{spl}})$ provides perfect translation indistinguishability. Since $\mathsf{QTI}_{\mathrm{spl}}$ has full support, we can conclude rd-indiff via Theorem 1.

Rd-indiff of **NewHope**. We next demonstrate how read-only indifferentiability can highlight subpar methods of oracle cloning, using the example of **NewHope** [2]. The base KEM KE_1 defined in the specification of **NewHope** relies on just two random oracles, G and H_4. (The base scheme defined by transform $\mathbf{T}_{10}$, which uses 3 random oracles H_2, H_3, and H_4, is equivalent to KE_1 and can be obtained by applying the output-splitting cloning functor to instantiate H_2 and H_3 with G. **NewHope**'s security proof explicitly claims this equivalence [2].)

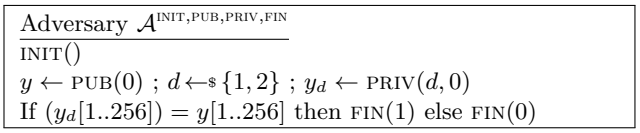

Fig. 7. Adversary against the rd-indiff security of $\mathbf{F}_{\texttt{NewHope}}$.

The final KEM KE_2 instantiates these two functions through SHAKE256 without explicit domain separation, setting $H_4(X) = \mathsf{SHAKE256}(X, 32)$ and $G_(X) = \mathsf{SHAKE256}(X, 96)$. For consistency with our results, which focus on sol function spaces, we model SHAKE256 as a random member of a sol function space SS with some very large output length L, and assume that the adversary does not request more than L bits of output from SHAKE256 in a single call. We let ES be the arity-2 sol function space defining sub-functions G and H_4. In this setting, the cloning functor $\mathbf{F}_{\texttt{NewHope}} : \mathsf{SS} \to \mathsf{ES}$ used by **NewHope** is defined by $\mathbf{F}_{\texttt{NewHope}}[s](1, X) = s(X)[1..256]$ and $\mathbf{F}_{\texttt{NewHope}}[s](2, X) = s(X)[1..768]$. We will show that this functor cannot achieve rd-indiff for the given oracle spaces and the working domain $\mathcal{W} = \{0,1\}^*$. In Figure 7, we give an adversary $\mathcal{A}$ which has high advantage in the rd-indiff game $\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F}_{\texttt{NewHope}},\mathsf{SS},\mathsf{ES},\mathcal{W},\mathsf{S}}$ for any indifferentiability simulator S. When $b = 1$ in game $\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F}_{\texttt{NewHope}},\mathsf{SS},\mathsf{ES},\mathcal{W},\mathsf{S}}$, we have that

$$y_d[1..256] = \mathbf{F}_{\texttt{NewHope}}[s](d, 0)[1..256] = s(0)[1..256] = y[1..256],$$

so adversary $\mathcal{A}$ will always call FIN on the bit 1 and win. When $b = 0$ in game $\mathbf{G}^{\text{rd-indiff}}_{\mathbf{F}_{\texttt{NewHope}},\mathsf{SS},\mathsf{ES},\mathcal{W},\mathsf{S}}$, the two strings $y_1 = e_0(1, X)$ and $y_2 = e_0(2, X)$ will have different 256-bit prefixes, except with probability $\epsilon = 2^{-256}$. Therefore, when $\mathcal{A}$ queries PUB(0), the simulator's response y can share the prefix of most one of the two strings y_1 and y_2. Its response must be independent of d, which is not chosen until after the query to PUB, so $\Pr[y[1..256] = y_d[1..256]] \leq 1/2 + \epsilon$, regardless of the behavior of S. Hence, $\mathcal{A}$ breaks the indifferentiability of $\mathbf{Q}^{\texttt{NewHope}}$ with probability roughly 1/2, rendering **NewHope**'s random oracle functor differentiable.

The implication of this result is that **NewHope**'s implementation differs noticeably from the model in which its security claims are set, even when SHAKE256 is assumed to be a random oracle. This admits the possibility of hash function collisions and other sources of vulnerability that are not eliminated by the security proof. To claim provable security for **NewHope**'s implementation, further justification is required to argue that these potential collisions are rare or unexploitable. We do not claim that an attack on read-only indifferentiability implies an attack on the IND-CCA security of **NewHope**, but it does highlight a gap that needs to be addressed. Read-only indifferentiability constitutes a useful tool for detecting such gaps and measuring the strength of various oracle cloning methods.

Acknowledgments

The authors were supported in part by NSF grant CNS-1717640 and a gift from Microsoft. Günther was additionally supported by Research Fellowship grant GU 1859/1-1 of the German Research Foundation (DFG).

References

1. M. Albrecht, C. Cid, K. G. Paterson, C. J. Tjhai, and M. Tomlinson. NTS-KEM. NIST PQC Round 2 Submission, 2019. 3, 14
2. E. Alkim, R. Avanzi, J. Bos, L. Ducas, A. de la Piedra, T. Pöppelmann, P. Schwabe, and D. Stebila. NewHope: Algorithm specifications and supporting documentation. NIST PQC Round 2 Submission, 2019. 3, 12, 26
3. N. Aragon, P. S. L. M. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Güneysu, C. Aguilar Melchor, R. Misoczki, E. Persichetti, N. Sendrier, J.-P. Tillich, V. Vasseur, and G. Zémor. BIKE: Bit flipping key encapsulation. NIST PQC Round 2 Submission, 2019. 3, 12
4. N. Aragon, O. Blazy, J.-C. Deneuville, P. Gaborit, A. Hauteville, O. Ruatta, J.-P. Tillich, and G. Zémor. LOCKER: Low rank parity check codes encryption. NIST PQC Round 1 Submission, 2017. 3, 12
5. R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé. CRYSTALS-Kyber: Algorithm specifications and supporting documentation. NIST PQC Round 2 Submission, 2019. 3, 14
6. H. Baan, S. Bhattacharya, S. Fluhrer, O. Garcia-Morchon, T. Laarohoven, R. Player, R. Rietman, M.-J. O. Saarinen, L. Tolhuizen, J. L. Torre-Arce, and Z. Zhang. Round5: KEM and PKE based on (ring) learning with rounding. NIST PQC Round 2 Submission, 2019. 3, 12
7. G. Banegas, P. S. L. M. Barreto, B. O. Boidje, P.-L. Cayrel, G. N. Dione, K. Gaj, C. T. Gueye, R. Haeussler, J. B. Klamti, O. N'diaye, D. T. Nguyen, E. Persichetti, and J. E. Ricardini. DAGS: Key encapsulation from dyadic GS codes. NIST PQC Round 1 Submission, 2017. 3, 10
8. M. Bardet, É. Barelli, O. Blazy, R. Canto-Torres, A. Couvreur, P. Gaborit, A. Otmani, N. Sendrier, and J.-P. Tillich. BIG QUAKE: Binary goppa quasi-cyclic key encapsulation. NIST PQC Round 1 Submission, 2017. 3, 10
9. M. Bellare, D. J. Bernstein, and S. Tessaro. Hash-function based PRFs: AMAC and its multi-user security. In M. Fischlin and J.-S. Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 566–595. Springer, Heidelberg, May 2016. 7
10. M. Bellare, H. Davis, and F. Günther. Separate your domains: NIST PQC KEMs, oracle cloning and read-only indifferentiability. Cryptology ePrint Archive, 2020. 6, 23
11. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In D. E. Denning, R. Pyle, R. Ganesan, R. S. Sandhu, and V. Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, Nov. 1993. 1, 7
12. M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Heidelberg, May / June 2006. 15
13. D. J. Bernstein, T. Chou, T. Lange, I. von Maurich, R. N. Rafael Misoczki, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, J. Szefer, and W. Wang. Classic McEliece: conservative code-based cryptography. NIST PQC Round 2 Submission, 2019. 3, 14

14. D. J. Bernstein, C. Chuengsatiansup, T. Lange, and C. van Vredendaal. NTRU Prime. NIST PQC Round 2 Submission, 2019. 3, 14
15. M.-S. Chen, A. Hülsing, J. Rijneveld, S. Samardjiska, and P. Schwabe. MQDSS specifications. NIST PQC Round 2 Submission, 2019. 14
16. J. H. Cheon, S. Park, J. Lee, D. Kim, Y. Song, S. Hong, D. Kim, J. Kim, S.-M. Hong, A. Yun, J. Kim, H. Park, E. Choi, K. Kim, J.-S. Kim, and J. Lee. Lizard public key encryption. NIST PQC Round 1 Submission, 2017. 3, 12, 13
17. J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård revisited: How to construct a hash function. In V. Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 430–448. Springer, Heidelberg, Aug. 2005. 4, 16
18. R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2003. 2
19. J.-P. D'Anvers, A. Karmakar, S. S. Roy, and F. Vercauteren. SABER: Mod-LWR based KEM. NIST PQC Round 2 Submission, 2019. 3, 12
20. G. Demay, P. Gaži, M. Hirt, and U. Maurer. Resource-restricted indifferentiability. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 664–683. Springer, Heidelberg, May 2013. 4, 16
21. A. W. Dent. A designer's guide to KEMs. In K. G. Paterson, editor, *9th IMA International Conference on Cryptography and Coding*, volume 2898 of *LNCS*, pages 133–151. Springer, Heidelberg, Dec. 2003. 2, 7, 8
22. O. Garcia-Morchon and Z. Zhang. Round2: KEM and PKE based on GLWR. NIST PQC Round 1 Submission, 2017. 3, 10
23. M. Hamburg. Post-quantum cryptography proposal: ThreeBears. NIST PQC Round 2 Submission, 2019. 3, 14
24. D. Hofheinz, K. Hövelmanns, and E. Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Y. Kalai and L. Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 341–371. Springer, Heidelberg, Nov. 2017. 2, 7, 8
25. A. Hülsing, J. Rijneveld, J. M. Schanck, and P. Schwabe. NTRU-HRSS-KEM: Algorithm specifications and supporting documentations. NIST PQC Round 1 Submission, 2017. 3, 14
26. D. Jao, R. Azarderakhsh, M. Campagna, C. Costello, L. D. Feo, B. hess, A. Jalali, B. Koziel, B. LaMacchia, P. Longa, M. Naehring, J. Renes, V. Soukharev, and D. Urbanik. Supersingular isogeny key encapsulation. NIST PQC Round 2 Submission, 2019. 3, 14
27. H. Jiang, Z. Zhang, L. Chen, H. Wang, and Z. Ma. IND-CCA-secure key encapsulation mechanism in the quantum random oracle model, revisited. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 96–125. Springer, Heidelberg, Aug. 2018. 2, 7, 8
28. X. Lu, Y. Liu, D. Jia, H. Xue, J. He, and Z. Zhang. LAC: Lattice-based cryptosystems. NIST PQC Round 2 Submission, 2019. 3, 12
29. U. M. Maurer, R. Renner, and C. Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In M. Naor, editor, *TCC 2004*, volume 2951 of *LNCS*, pages 21–39. Springer, Heidelberg, Feb. 2004. 2, 4, 16, 18
30. C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, A. Hauteville, O. Ruatta, J.-P. Tillich, and G. Zémor. ROLLO: Rank-ouroboros, LAKE, & LOCKER. NIST PQC Round 2 Submission, 2018. 3, 12, 13

31. C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, and G. Zémor. Rank quasi-cyclic (RQC). NIST PQC Round 2 Submission, 2019. 3, 14
32. C. A. Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. D. P. Gaborit, and E. P. G. Zémor. Hamming quasi-cyclic (HQC). NIST PQC Round 2 Submission, 2019. 3, 14
33. A. Mittelbach. Salvaging indifferentiability in a multi-stage setting. In P. Q. Nguyen and E. Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 603–621. Springer, Heidelberg, May 2014. 4, 16
34. M. Naehrig, E. Alkim, J. W. Bos, L. Ducas, K. Easterbrook, B. LaMacchia, P. Longa, I. Mironov, V. Nikolaenko, C. Peikert, A. Raghunathan, and D. Stebila. FrodoKEM: Learning with errors key encapsulation. NIST PQC Round 2 Submission, 2019. 3, 14
35. NIST. Post-Quantum Cryptography Standardization Process. https://csrc.nist.gov/projects/post-quantum-cryptography. 2
36. NIST. Federal Information Processing Standard 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, Aug 2015. 2
37. NIST. PQC Standardization Process: Second Round Candidate Announcement. https://csrc.nist.gov/news/2019/pqc-standardization-process-2nd-round-candidates, Jan. 2019. 2
38. T. Plantard. Odd manhattan's algorithm specifications and supporting documentation. NIST PQC Round 1 Submission, 2017. 3, 12
39. T. Ristenpart, H. Shacham, and T. Shrimpton. Careful with composition: Limitations of the indifferentiability framework. In K. G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 487–506. Springer, Heidelberg, May 2011. 4, 6, 16, 18, 22
40. T. Saito, K. Xagawa, and T. Yamakawa. Tightly-secure key-encapsulation mechanism in the quantum random oracle model. In J. B. Nielsen and V. Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 520–551. Springer, Heidelberg, Apr. / May 2018. 2, 7, 8
41. M. Seo, J. H. Park, D. H. Lee, S. Kim, and S.-J. Lee. Proposal for NIST post-quantum cryptography standard: EMBLEM and R.EMBLEM. NIST PQC Round 1 Submission, 2017. 3, 14
42. N. P. Smart, M. R. Albrecht, Y. Lindell, E. Orsini, V. Osheter, K. G. Paterson, and G. Peer. LIMA: A PQC encryption scheme. NIST PQC Round 1 Submission, 2017. 3, 14
43. R. Steinfeld, A. Sakzad, and R. K. Zhao. Titanium: Proposal for a NIST post-quantum public-key encryption and KEM standard. NIST PQC Round 1 Submission, 2017. 3, 12
44. Y. Zhao, Z. Jin, B. Gong, and G. Sui. A modular and systematic approach to key establishment and public-key encryption based on LWE and its variants. NIST PQC Round 1 Submission, 2017. 3, 12