Fujisaki-Okamoto Transform and Applications to Supersingular Isogeny Diffie Hellman Lecture Notes

Chelsea Komlo

April 16, 2020

1 Introduction

In this set of lecture notes, our goal is to understand SIKE, the set of transformations that distinguish SIDH from SIKE, and importantly, why these transformations are necessary in the first place. Towards this goal, we'll first review SIDH, and then look at an attack on SIDH by Galbraith. We'll then look at the FO transform, a generalized transformation for public-key encryption schemes which has been used in conjunction with other post-quantum schemes. We'll then see how the FO transform can be applied to SIDH, resulting in SIKE, and how the application of this transform results in SIKE as an IND-CCA2 scheme. We'll also discuss how the application of the FO transform to SIKE protects against Galbraith's attack specifically.

2 SIDH Recap

Let's quickly review SIDH, as we'll need this for understanding the attack. In this setting, we assume Alice is the initiator of a SIDH key exchange with Bob; we assume Alice builds two-degree isogenies and Bob builds three-degree isogenies.

2.1 Preliminaries

Let p be a prime of the form $2^{e_1}3^{e_2}-1$. Let (sk_ℓ, pk_ℓ) be a supersingular isogeny key pair, where the secret key sk_ℓ is a secret scalar drawn from a keyspace K_ℓ, where $\ell \in \{2,3\}$. Let K_2 be the set of integers in the range $\{0,\ldots,2^{e_1}-1\}$ and K_3 be the set of integers in the range $\{0,\ldots,3^{e_2}-1\}$.

Let the public key pk_ℓ be the curve $E/\langle P+[sk_\ell]Q\rangle$, where P,Q are publicly-known points of order either 2^{e_1} (when $\ell=2$) or 3^{e_2} (when $\ell=3$) on E.

2.2 SIDH Key Exchange

Alice begins by performing the following steps:

1. Randomly sample a secret scalar $n_a \in K_2$; use this secret scalar to generate the kernel defining her secret isogeny $\phi_a : E \to E_a = E/\langle P_a + [n_a]Q_a \rangle$
2. Derive the codomain curve serving as her ephemeral public key, along with Bob's auxiliary points after "pushing" them through to E_a by applying ϕ_a; in total, her public key is these three values $(E_a, \phi_a(P_b), \phi_a(Q_b)$, which she then sends to Bob.

After receiving Alice's ephemeral public key, Bob performs the following steps:

1. Randomly sample a secret scalar $n_b \in K_3$, computes the kernel defining his secret isogeny $\phi_B : E \to E_B = E/\langle P_B + [n_B]Q_B \rangle$
2. Derive his public key $E_B, \phi_B(P_A), \phi_B(Q_A)$, which he then sends to Alice.
3. Bob computes their shared secret by calculating the codomain curve:

$$E_A/\langle \phi_A(P_B) + [n_B]\phi_A(Q_B) \rangle$$

After receiving Bob's public key, Alice completes the key exchange deriving their shared secret, performing the following steps:

$$E_B/\langle \phi_B(P_A) + [n_A]\phi_B(Q_A) \rangle$$

3 Galbraith attack on Static/Ephemeral SIDH

We'll now discuss an attack presented by Galbraith et al [2] on SIDH in the setting where Bob maintains a static keypair but Alice is allowed to use an ephemeral keypair.

The attack is defined by the below properties:

- **Active attack against static-key SIDH.** The attack involves an active adversary that is allowed to sample an ephemeral key for each exchange, while the victim maintains a static key.
- **The attacker uses invalid public keys to perform SIDH, and tests the resulting shared secret.** Because there is no good mechanism to validate public keys in SIDH, the victim cannot detect the attack.
- **Allows for recovering the entire static key bit-by-bit.** Each query leaks one bit of information about the victim's private key.

Figure 1: Galbraith Attack on Static/Ephemeral SIDH

Step 1:

Step 1a: Honestly follow the SIDH protocol, resulting in her ephemeral public key $E_A, \phi_A(P_B), \phi_A(Q_B)$, and her shared secret with Bob:

$$E_{AB} = E_B / \langle \phi_A(P_B), [a_2]\phi_A(Q_B) \rangle$$

Step 1b. Alice then tests the j-invariant j and records it.

Step 2: Alice then performs another key-exchange with Bob, but instead of sending a fresh ephemeral public key, she sends:

$$E_A, \phi_A(P_B), \phi_A(Q_B) + [2^{n-1}\phi_A(P_A)]$$

Step 3. Bob performs the key exchange, sending Alice his static public key $E_B, \phi_B(P_A), \phi_B(Q_A)$

Step 4. Alice then tests the resulting j-invariant. If it is the same as in Step 1, the last bit of Alice's secret is even, otherwise odd.

3.1 Attack Description

We'll first describe how the attack can extract the first bit of the victim's secret key, and then describe how this technique can be expanded to extract the remaining bits.

3.1.1 Attack to Extract the First Bit of the Victim's Secret.

Let Bob be the party maintaining a static key, and Alice be the attacker who initiates key exchange using her ephemeral key. We'll now describe the steps that Alice takes to perform the attack on her victim, Bob.

Attack Intuition. When Alice submits her maliciously-crafted public key, the resulting shared secret curve is:

$$E_{AB'} = E_B / \langle \phi_A(P_B) + [a_2](\phi_A(Q_B) + [2^{n-1}]\phi_A(P_B)) \rangle$$

This value expands to:

$$E_{AB'} = E_B / \langle \phi_A(P_B) + [a_2]\phi_A(Q_B) + [a_2][2^{n-1}]\phi_A(P_B) \rangle$$

The attack relies on Alice determining whether or not E'_{AB} is isomorphic to E_{AB}; when it is, the last bit of Bob's secret is 0, otherwise the last bit must be 1. However, why does Alice learning this information leak this single bit of information?

Recall that $\phi_A(P_B), \phi_B(Q_B) \in E_A[2^n]$ (e.g, the order of $\phi_A(P_B), \phi_B(Q_B)$ is 2^n) . Now, let's examine the last term in E'_{AB} (recall this is the resulting shared secret *after* Alice has performed her attack), the value $[a_2][2^{n-1}]\phi_A(P_B)$, and how this term behaves depending on the value of Bob's secret scalar a_2.

When a_2 is even, $[a_2][2^{n-1}]\phi_A(P_B)) = 0$. We know this is true because $\phi_A(P_B)$ has order 2^n, and if a_2 is even, then the term $a_2 \cdot 2^{n-1}$ will be larger than 2^n (but still even), and so this entire term will cancel.

Consequently, when $[a_2][2^{n-1}]\phi_A(P_B)) = 0$, then the j-invariant of E'_{AB}—the resulting shared secret of the key exchange when Alice submitted her maliciously-crafted public key—will be the same as E_{AB}, the shared secret where Alice followed the protocol honestly. However, when a_2 is odd, then $[a_2][2^{n-1}]\phi_A(P_B)) \neq 0$, and then the j-invariant of $E_{AB'}$ will not be equal to the j-invariant of E_{AB} (in other words, these curves are not equal up to isomorphism).

3.1.2 Finding the remaining terms in the secret

A similar approach to the above can be used to determine the remaining bits of Bob's static private key a_2. Notably, the attack to determine each bit makes use of the fact that $\phi_A(P_B), \phi_A(Q_B)$ have order 2^n, and submit crafted public keys to iteratively test each bit, continuing to test whether the key exchange completes successfully or not as the oracle.

4 Fujisaki-Okamoto (FO) Transform

Let's now discuss what the Fujisaki-Okamoto (FO) transform is, and why it is useful to prevent attacks such as Galbraith's attack that was just discussed. We will also review how the FO transform applies to KEMs such as SIKE.

4.1 FO Overview

To understand the FO transform, we need to first look at security models and definitions. We'll then explain how FO defines a transform to improve a weakly-secure public-key encryption scheme against a passive adversary to strongly-secure against an active adversary.

4.1.1 Security Models and Definitions

Random Oracle Model. Note that the original FO transform and later similar techniques (that we describe later) rely on the security of the *Random Oracle Model*. In other words, the random oracle model in a proof of security assumes that the output from a cryptographic hash function is indistinguishable to an

adversary from that of a random oracle regardless of the input into the hash function.

OW-CPA: One-Way Chosen Plaintext Attack. The adversary in an OW-CPA model is *passive*. In this model, the adversary submits two messages (m_0, m_1) to the challenger. Using the public key, the challenger encrypts $c_B : B \in \{0, 1\}$, and returns c_B to the adversary. The adversary wins if it outputs the correct guess for B. In this setting, the adversary has access only to an encryption oracle and the public key, as in a public-key setting, anyone can encrypt to a public key.

IND-CCA2: Indistinguishability under Chosen Ciphertext Attack The IND-CCA2 model is the same as OW-CPA, but in this setting the adversary is given additional powers. The adversary in an IND-CCA2 model is *adaptive*; this means that the adversary can continue querying oracles even after receiving the challenge ciphertext. In this mode, the adversary has access to both an encryption and decryption oracle, and can submit any message/ciphertext to these oracles *except* for the challenge ciphertext.

4.1.2 FO Transformation: OW-CPA to IND-CCA2

The FO transform is a general mechanism that transforms an OW-CPA public-key cryptosystem to an IND-CCA2 cryptosystem by applying specific techniques to the encryption and decryption functions of the scheme.

This transformation mechanism was originally introduced by Fujisaki and Okomoto in their seminal paper [1] with a specific construction. However, the term "FO transform" now is used more generally as a set of techniques to achieve IND-CCA2 security in combination with an OW-CPA public-key encryption scheme.

Intuitively, the FO transform ensures that—after applying the set of transformations to the public-key cryposystem—an adversary working to guess the contents of a ciphertext c cannot do so without direct knowledge of the corresponding secret message m for which c is the encryption of, even if the adversary has knowledge of an arbitrary number other m', c' pairs of their choosing.

4.2 FO when applied to Key Encapsulation Mechanism (KEM)

Hofheinz, Hovelmanns, and Kiltz [3] present a toolkit of composable transformations to turn OW-CPA public-key cryptosystems into those achieving IND-CCA2 security using a KEM API. While their techniques have similarities to the original FO transform presented by Fujisaki and Okomoto, the authors specifically present transformations in modular steps to allow for composability of techniques, which allows for different instantiations and proofs of security for each transform.

Application to SIKE. While Hofheinz, Hovelmanns, and Kiltz present a range of transformations, we review only those which are employed in the

Supersingular Isogeny Key Exchange (SIKE) protocol, which we further describe in Section 5.

4.2.1 Definitions

Let's first look at several additional definitions of security models that are used for proofs of security in Hofheinz, Hovelmanns, and Kiltz.

Plaintext Checking Oracle. The plaintext checking oracle accepts as input a message m and ciphertext c, and outputs a boolean indicating if the decryption of c using a secret key sk results in m.

Ciphertext Checking Oracle. The ciphertext validity checking oracle accepts as input a ciphertext c that is *not* the challenge ciphertext, and outputs a boolean indicating if the decryption of c results in a valid message.

One-Way under Plaintext and Validity Checking Attacks (OW-PCVA). This security definition is introduced by the authors for a transformation that is used as "stepping stone" in a set of composable transformations, allowing the complete set to bridge OW-CPA to OW-PCVA to IND-CCA2. This definition allows the adversary access to a plaintext checking oracle and a ciphertext validity oracle.

Next, we'll look at the transformations presented in Hofheinz, Hovelmanns, and Kiltz that are used in SIKE.

4.2.2 Transformation T- "Derandomization" plus "Re-encryption"

This first transformation requires an OW-CPA public-key cryptosystem and a cryptographically-strong hash function G, and produces an OW-PVCA cryptosystem. This transformation works within the random oracle model, assuming the output of G is indistinguishable to a polynomial-time adversary from that of a perfectly random oracle.

The T transformation is presented in Algorithm 1 and 2.

Algorithm 1 T transformation Enc_1

Input pk, m
Output c

1: $c := Enc(pk, m, G(m))$
2: return c

In the equation for Enc_1, $G(m)$ acts as the random coin into Enc, ensuring that the output of Enc is *deterministic*. As such, every input m to Enc_1 for a specific public key pk has a unique corresponding output (assuming finding a hash collision for G is hard).

In the equation for Dec_1, the use of $Enc(sk, m', G(m')) \neq c$ provides a *re-encryption* step that uses the target message m' to determine if encryption under the secret key corresponds to the target ciphertext; if not, return $\perp$.

Proof Intuition. At a high level, the proof strategy demonstrates that the T transformation *perfectly simulates* the plaintext-checking and ciphertext va-

Algorithm 2 T transformation Dec_1

Input sk, c
Output m

1: $m' = Dec(sk, c)$
2: **if** $m' = \perp$ **or** $Enc(sk, m', G(m')) \neq c$ **then**
3: return $\perp$
4: **else**
5: return m'
6: **end if**

lidity oracles to an adversary in an OW-PCVA setting *without* requiring knowledge of the corresponding private key. Consequently, an adversary with access to an OW-CPA public-key encryption scheme that implements T oracles does not gain additional powers when moving from the OW-CPA game to the OW-PCVA game.

Let's now see how the plaintext-checking oracle and the ciphertext validity checking oracle can be simulated without knowledge of the secret key sk in Algorithms 1 and 2. Recall that when simulating oracles to an adversary, the simulator will also see any input queried by the adversary to the oracle, including case queries to G. By observing the adversary's inputs to G, the simulator can simulate the plaintext-checking oracle and the ciphertext validity checking oracle on line 2 in Algorithm 2, and consequently can perfectly simulate Dec using a secret key chosen at random, having recorded all inputs to Enc and G.

In short, because Enc is deterministic, the plaintext-checking oracle can be perfectly simulated. Further, the ciphertext validity checking oracle can be perfectly simulated by the re-encryption step on line 2, which implicitly checks the validity of the ciphertext.

4.2.3 Transformation $U^{\not\perp}$- "Hashing"

The $U^{\not\perp}$ transformation requires as input an OW-PCVA public-key cryptosystem and a cryptographically-strong hash function H, and produces an IND-CCA2 adaptively secure key encapsulation mechanism (KEM). This transformation works also within the random oracle model, and can be composed with the T transformation discussed above (such that resulting transformation for a OW-CPA public-key encryption scheme PKE is $U^{\not\perp}(T(\text{PKE}, G), H)$).

As seen in Algorithm 3 and 4, the $U^{\not\perp}$ transformation uses a cryptographic hash function H to generate the shared secret K as the hash of the secret message along with the provided ciphertext, assuming that m was successfully decrypted. Doing so "binds" the message to the ciphertext, such that any change in either m or c produces an entirely different secret K.

Implicit Rejection. The $U^{\not\perp}$ transformation uses an "implicit rejection" mechanism; in the case that the decapsulate function fails to produce a valid message to after decrypting the ciphertext submitted as input, the decapsulate function instead derives a secret key K to use with a locally-selected secret

Algorithm 3 $U^{\not\perp}$ transformation Encaps

Input $pk, (c_0, c_1)$
Output K, c

1: $m \xleftarrow{\$} M$
2: $c \leftarrow Enc_1(ok, m)$
3: $K := H(m, c)$
4: return (K, c)

Algorithm 4 $U^{\not\perp}$ transformation Decaps

Input $(sk, s), c, (c_0, c_1)$
Output K

1: $m' := Dec_1(sk, c)$
2: **if** $m' \neq \perp$ **then**
3: return $K := H(m', c)$
4: **else**
5: return $K := H(s, c)$
6: **end if**

value, in order to not leak information to the adversary about the reason for the key exchange failure.

Proof intuition. The proof strategy here is to simulate the decapsulation oracle to the adversary without knowledge of the secret key. In doing so, the proof demonstrates that an adversary in the IND-CCA2 setting with access to the decapsulation oracle for any message, ciphertext pair *except* for the challenge ciphertext is no more powerful than an adversary in the OW-PCVA game.

Recall that the simulator can view all queries the adversary makes to any of the oracles, including the random oracles (in this case simulated by G and H). By observing the adversary queries to G, H, and *Encaps* (and recording the outputs), the simulator can perfectly simulate *Decaps* to the adversary without knowledge of the secret key sk. Consequently, an adversary that has arbitrary access the *Decaps* oracle can query any message, ciphertext pair that is *not* the challenge ciphrtext, but will not gain any special knowledge over an adversary without access to *Decaps*.

Another way to think about the transform to IND-CCA2 security of this transformation is to consider how knowledge of message, ciphertext (m, c) pairs impacts the adversary's ability to gain information about the challenge ciphertext c'. Examining line 3 in *Decaps*, the output from H depends on its input (m, c). If the adversary knows only challenge ciphertext c' and a set of m, c other pairs, the adversary must *also* know m' in order to derive the shared secret $K := H(m', c')$.

5 SIKE- Supersingular Isogeny Key Encapsulation

SIKE (Supersingular Isogeny Key Encapsulation) [4] is a submission by Jao et al. to the NIST competition to standardize post-quantum key exchange protocols. SIKE is an adaptation of the SIDH key exchange protocol described in Section 2, and is defined as a KEM. Further, SIKE instantiates the $U^{\not\perp}$ and T transformations described in Section 4.2.

We'll first look at the core of the SIKE exchange, and then discuss how SIKE instantiates these FO transformations and in doing so, protects against the Galbraith attack discussed in Section 3.

5.1 SIKE Protocol

Additional Preliminaries. Let M be the message space $\{0,1\}^n$. Let $isogen_\ell(sk_\ell \rightarrow pk_\ell)$ be a function that generates a supersingular key pair where the secret key is from the keyspace K_2 or K_3 (as described in Section 2). Let $isoex(pk_2, sk_3) \rightarrow j$ be a supersingular isogeny key exchange function whose output is the shared secret j-invariant.

Let's continue using Alice as the key exchange initiator (whose keypairs are denoted (sk_2, pk_2)) who uses a fresh ephemeral key each time, and Bob as the party who maintains a static keypair denoted (sk_3, pk_3).

5.1.1 Public Key Encryption

Below, we only define the functions *Enc* and *Dec* for public-key encryption; see the full SIKE specification [4] for the *Gen* function definition which produces an isogeny keypair (further described in Section 2).

Algorithm 5 Enc

Input $pk_3, m \in M, r \in \mathcal{K}_2$
Output (c_0, c_1)

1: $sk_2 \leftarrow r$
2: $c_0 \leftarrow isogen_2(sk_2)$
3: $j \leftarrow isoex_2(pk_3, sk_2)$
4: $h \leftarrow F(j)$
5: $c_1 \leftarrow h \oplus m$
6: **return** (c_0, c_1)

Key Generation. In this setting, *Gen* simply generates a supersingular isogeny keypair using $isogen_\ell$, and additionally generates a secret value $s \leftarrow_\$ \{0,1\}^n$, which is used for implicit rejection, as we'll later see in *Decaps*.

Encryption. Encrypting to a public key as specified in Algorithm 5 involves generating a fresh keypair using *isogen*, performing a key exchange with Bob using *isoex*, and using the shared secret j-invariant to symmetrically encrypt

Algorithm 6 Dec

Input $sk_3, (c_0, c_1)$
Output m

1: $j \leftarrow isoex_2(c_0, sk_3)$
2: $h \leftarrow F(j)$
3: $m \leftarrow h \oplus c_1$
4: **return** m

the message. Note the symmetric encryption step, which is a simple xor of the hash of the j-invariant to the message.

Decryption. In order to decrypt the secret message using ciphertexts (c_0, c_1), the decryption function as specified in Algorithm 6 must be able to successfully perform $isoex$ in reverse; using the supplied ephemeral public key c_0 in conjunction with their own static secret key sk_3 to arrive at the shared secret j-invariant, which is then used to retrieve the message by finding the xor of this value with the supplied ciphertext (serving as a one-time pad).

5.2 Instantiation of FO

We'll now first discuss how SIKE implements the T and $U^{\not\perp}$ transformations described in Section 4.2, and then discuss how these transformations prevent Galbraith's attack.

Algorithm 7 Encaps

Input pk_3
Output (c, K)

1: $m \xleftarrow{\$} \{0,1\}^n$
2: $r \leftarrow G(m \| pk_3)$
3: $(c_0, c_1 \leftarrow Enc(pk_3, m, r)$
4: $K \leftarrow H(m \| (c_0, c_1))$
5: **return** $(c_0, c_1), K$

5.2.1 Instantiation of Transform T.

We now describe how SIKE implements the "Derandomization" plus "Re-encryption" transform described in Section 4.2.2 in its instantiation of *Encaps* (described in Algorithm 7 and *Decaps* (described in Algorithm 8).

Derandomization. The output of *Enc* is always deterministic because instead of generating a random value to use as the secret key for $isogen$ (line 3), it instead receives this value as random coins into the function. This random value is provided by *Encaps*, which produces it by randomly sampling a message m and then hashing m and the receiving party's public key (in this case, Bob's static key).

Algorithm 8 Decaps

Input $(s, sk_3, pk_3), (c_0, c_1)$
Output K

1: $m' \leftarrow Dec(sk_3, (c_0, c_1))$
2: $r' \leftarrow G(m' \| pk_3)$
3: $c_0' \leftarrow isogen(r')$
4: **if** $c_0' = c_0$ **then**
5: $\quad K \leftarrow H(m' \| (c_0, c_1))$
6: **else**
7: $\quad K \leftarrow H(s \| (c_0, c_1))$
8: **end if**

Re-encryption. The re-encryption step is implemented in the *Decaps* step (line 3 in Algorithm 8), where after decrypting the secret message, the protocol performs the re-encryption step performing *Enc* using the message that was obtained during the decryption step in line 1. If the derived output matches the ciphertext received as input (which is also Alice's ephemeral public key), the re-encryption is successful and *Decaps* proceeds to derive the shared secret. Otherwise, the function proceeds as described above in the "implicit rejection" step, deriving the key from a locally-generated secret.

5.2.2 Instantiation of Transform $U^{\not\perp}$.

Hashing. SIKE implements the "Hashing" transform in the *Decaps* step presented in Algorithm 8. If the *isogen* function with the provided secret $G(m'||pk_3)$ produces the correct ephemeral public key c_0' for Alice (line 3), then the shared secret is simply the resulting message hashed with the ciphertexts provided by Alice (line 5). As SIKE uses the implicit rejection variant of the transform, the case of failing to reproduce the correct ciphertext results in generating K as the hash of the locally-derived secret s along with the provided ciphertexts (line 7).

5.3 Analysis of Security

Now that we've seen how SIKE implements the T and $U^{\not\perp}$ transformations from Hofheinz, Hovelmanns, and Kiltz [3], let's now discuss how these transforms protect against the Galbraith attack discussed in Section 3.

Protection against Galbraith's Attack: Intuition. At a high level, the implementation of the FO transformation in SIKE ensure that a key exchange initiator Alice must commit to a *valid* public/private keypair before performing the key exchange, and that the public key she sends to Bob must be this valid public key. If Alice sends a maliciously-crafted public key, the key exchange will fail in such a way that will not leak the reason of failure to Alice (due to implicit rejection). In short, applying the FO transform to SIDH allows for *indirect public key validation*, even in the context of key agreement [5].

Let's look at this a bit more in detail. Say Alice is attempting to conduct the Galbraith attack, but in the context of SIKE. She begins by performing a SIKE key exchange honestly, similar to step 1 in the original attack (described in Figure 1). Next, she performs step 2 by generating an invalid public key using values obtained during step 1; this would result in an invalid c_0 value sent to Bob.

However, when Bob performs *Decaps*, the check on line 4 in Algorithm 8 will prevent the Galbraith attack directly, as Bob will not arrive at the same c_0 value as submitted by Alice. We know this to be the case because Alice herself does not know the secret key corresponding to her maliciously-crafted c_0 (due to the assumption that the hash function G can be modeled as a random oracle). Hence, when Bob attempts to decrypt using c_0, c_1 and then perform re-encryption by performing *isogen* (line 3 of *Decaps*), he will arrive at a different c_0 value than submitted by Alice.

As such, the transformations protect against the adaptive attack by Galbraith.

References

[1] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26:80–101, 2011.

[2] Steven D. Galbraith, Christophe Petit, Barak Shani, and Yan Bo Ti. On the security of supersingular isogeny cryptosystems. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 63–91, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[3] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the fujisaki-okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography*, pages 341–371, Cham, 2017. Springer International Publishing.

[4] David Jao. Supersingular Isogeny Key Exchange. https://sike.org, 2019. last accessed 2020-03-19.

[5] Daniel Kirkwood, Bradley C. Lackey, John McVey, Mark Motley, Jerome A. Solinas, and David Tuller. Failure is not an Option: Standardization Issues for Post-Quantum Key Agreement. https://csrc.nist.gov/csrc/media/events/workshop-on-cybersecurity-in-a-post-quantum-world/documents/presentations/session7-motley-mark.pdf, 2015. last accessed 2020-03-27.