DRAFT December 13, 2024

Structured Data Folding with Transmutations (SDFT): A Framework for Cryptographic Agility

Yoon Ho Auh
NUTS Technologies, Inc.
Illinois, USA
yoon@nutstechnologies.com

Nick Bennig
NUTS Technologies, Inc.
Wisconsin, USA
nick@nutstechnologies.com

Sotirios Triantafillou
NUTS Technologies, Inc.
Florida, USA
soto@nutstechnologies.com

COL (Ret) Robert Banks, Ph.D.
Washington D.C., USA
rebanks83@gmail.com

Eric Hauk
NUTS Technologies, Inc.
Illinois, USA
eric@nutstechnologies.com

Yoonil Auh
Kyung Hee (Cyber) University
Seoul, South Korea
yoonilauh@gmail.com

John O'Neill
NUTS Technologies, Inc.
New Hampshire, USA
john@nutstechnologies.com

Jin Auh
NUTS Technologies, Inc.
Illinois, USA
jin@nutstechnologies.com

Gilgamesh Auh
NUTS Technologies, Inc.
Illinois, USA
jin@nutstechnologies.com

Vasili Apostolopoulos
NUTS Technologies Inc.
Illinois, USA
vasili@nutstechnologies.com

Abstract

Structured Data Folding with Transmutations (SDFT) provides a framework for producing portable, secure messages enabling facile cryptography transitions indefinitely. As ciphers age and exhibit fallibility, new ciphers are introduced resulting in a parade of cryptography transitions over time. SDFT streamlines cipher transitions at the message level for both Data-At-Rest (DAR) and Data-In-Transit (DIT), unifying these traditionally bifurcated sub-fields of applied cryptography. SDFT advances beyond concepts of Message Level Security (MLS) by allowing every SDFT message to self-describe and self-prescribe how the message within was produced from an application memory object. Cipher transitions are introduced with various levels of urgency by each organization. Custom cryptography implementations will incur high transition costs due to a shortage of qualified experts. Indifference or ignorance can lead to catastrophic data breaches as adversaries resort to retrospective decryption attacks. Even the best cryptography transitions may introduce new flaws into the application, leading to weaker security than expected. This project researched and integrated several available Python Post-Quantum Cryptography (PQC) modules into the existing SDFT Python library to handle the latest NIST PQC standardized and candidate algorithms. The results show how the SDFT framework seamlessly integrates the newest cryptographic algorithms, while maintaining backwards compatibility with legacy cryptographic algorithms by introducing a simple, text-based sequence of commands embedded alongside the message itself using a technique we call data folding. Cryptographic agility, or cryptoagility, at this message level reveals a new feature in a single communication session, the capability to engage in negotiation-less cryptographic communications, where each side can use their choice of cryptography to encrypt their messages. The SDFT framework is a new technical approach to how cryptoagility can be systematically provided at the lowest layers of data protection in a universal way and presents a strong candidate for a standard in retail, commercial, and government applications of cryptographic algorithms and technologies.

1. Introduction

On August 13, 2024, NIST standardized three Post-Quantum Cryptography (PQC) algorithms [1] which are resistant to the capabilities of cryptanalytically relevant quantum computers (CRQC). Sufficiently powerful quantum computers capable of applying Shor's or Grover's algorithms to break today's best cryptographic algorithms are not yet known to exist but at the current rate of progress, it is estimated to be a national threat by 2035 prompting the NSA to publish guidelines in the CNSA 2.0 publication [2] for all Defense and Intelligence systems to be PQC compliant by 2035. The White House NSM-10 [3] directs a whole-of-government and whole-of-society strategy towards effectively protecting against the CRQC threat. The U.S. Congress passed the Quantum Computing Cybersecurity Preparedness Act [4] turning the PQC transition effort into law for all Federal systems. Efforts like the NCCoE MPQC project [5] (National Cybersecurity Center of Excellence Migration to PQC) act as the ringing of the village bell tower, alerting everyone in the vicinity of the implications of not being prepared for this transition and providing helpful roadmaps and tools which can form the basis of a systematic action plan. The transition to PQC is not the first time cryptography transitions have occurred in the field of applied cryptography; in fact, transitions in cryptographic primitives have occurred consistently over time (i.e. DES, 3DES, AES, RSA, etc.) [6].

The pace of cryptographic transitions is anticipated to accelerate with the continuous growth of computational power and the increasing sophistication of cryptographic attack methods. These advancements will gradually undermine even the most advanced cryptographic systems currently in use. While the recently standardized Post-Quantum Cryptography (PQC) algorithms represent a significant step forward, they may not be a permanent solution but part of an ongoing evolution in cryptographic defenses. As new vulnerabilities emerge, further transitions will inevitably be required.

The prospect of Cryptanalytically Relevant Quantum Computers (CRQCs) sets a clear expiration date for widely used asymmetric ciphers such as RSA and Elliptic Curve Cryptography (ECC). These algorithms, which have served as the cornerstone of secure communications for decades, are on the verge of obsolescence due to the looming threat of CRQC decryption capabilities. This transition could exceed the scale and complexity of the Y2K challenge from a quarter-century ago, as RSA and ECC are fundamental to nearly all digital interactions, including secure browsing, e-commerce, national security communications, banking, and cryptocurrency transactions. NIST states that most systems we have today are not cryptographically agile and that this transition can take up to a decade or two [7] [8] [9]. There are several factors complicating this impending transition to PQC:

1. There are not enough applied cryptographers to make all the necessary changes to all the important systems.

2. There will be many PQC algorithms replacing the two most widely used ciphers: ECC and RSA. In August 2024, NIST standardized 3 PQC algorithms, with 4 more currently in the evaluation queue targeting the end of 2025. Further, the NIST has another suite of 14 PQC candidates they are examining beyond that time frame [10]. The transitions might be more frequent than one might expect.

3. Transitions to PQC may come in two distinct forms: Data-At-Rest (DAR) and Data-In-Transit (DIT). These may take different forms for different systems. Whereas DIT encryption is ephemeral and can be reset on a per-session basis, DAR presents a more complicated set of predicaments due to long term storage, archiving, volume of encrypted data, key management, backwards compatibility, and availability concerns. In short, issues related to DAR requires a reimagining of how to approach the encryption of data for both DAR and DIT, especially in light of "store now, decrypt later" type of attacks. An ideal solution is one where encryption can be applied in a consistent manner for both modes of data.

Our research provides a path to performing a single transition to a framework which supports all future cryptography transitions as a trivial process by viewing the issue as algorithm normalization and management rather than manual algorithm transitions. We implemented a proof-of-concept Python library module to deliver the NIST PQC algorithms in a new secure messaging protocol framework, called Structured Data Folding with Transmutations (SDFT). SDFT eases the recurring cryptography transition issues going forward in systems incorporating cryptography. SDFT integration will require one transition to SDFT; thereon, all future cryptography transitions may be trivial.

2. Innovation: SDFT

eNcrypted Userdata Transit & Storage (NUTS) [11] is an integrated set of technologies developed by NUTS Technologies Inc. (NUTSTECH) researching and building products for data privacy and data management. **Structured Data Folding with Transmutations** (SDFT) is a specialty research area of NUTSTECH and which enables the construction of secure encapsulations called nut capsules for NUTS.

Traditionally, applied cryptography has been delivered with a craftsman's approach: most cryptographic systems are custom built by experts, and they are generally incompatible with one another unless great effort is involved. Integrations typically require further custom modifications by experts. There are standards for message protocols such as HTTPS and SSL/TLS, but there are no universal standards for encrypted persistent data storage methods and formats, ***which anticipate future cryptography transitions on a per-message basis***. SDFT drills down ***cryptoagility*** down to the per-message basis to offer maximum flexibility and granularity. Further, there are no known protocols which can be used for both DAR and DIT in a consistent way.

The SDFT approach is a simple concept to understand once explained. Our initiative and motivations behind developing SDFT were to advance our data-centric security research by allowing higher abstractions of applying cryptography to complex secure data structures without being dragged down by the myriad precise implementation details required in the application of cryptography. A project like NUTS demanded better ways to package applied cryptography in an organized, modular, and scalable way for both DAR and DIT.

SDFT was developed using a design thinking methodology [12] in answering the questions: "who will be using the product?" and "how will this solution impact the user?" Programmers and applied cryptographers will be using SDFT. SDFT's impacts are to provide a consistent, repeatable, and scalable method of applying cryptography on a broad scale, and to rapidly expand the base of developers to perform cryptography transitions by lowering the developers' experience baseline significantly without compromising security and quality.

The features of SDFT are tightly integrated in a comprehensive framework following the Transmutations Organizing Principles (TOP) [13] to provide streamlining in the application of cryptography in producing ciphered data. The applied cryptographic field has some efforts at streamlining

and structuring such as JWK (IETF Java Web Encryption data structures [14]) and others, but the **comprehensiveness** and **coverage** of SDFT technology marks a drastic departure from previous efforts by balancing freedom of choice of cryptography, ease of application of cryptography, and transitions to new cryptography. Comprehensiveness [15] and Coverage [16] are the requirements by different authors for the Next Generation of Security [17]. The proper application of cryptography involves a combination of knowledge in applied cryptography, in-memory data object designs, and message protocols. The SDFT framework incorporates all these distinct and highly technical specialties into a single cohesive toolset thereby filling a growing technological gap that the industry is rapidly becoming aware of called crypto-agility [18].

2.1. The Assembly Line

Henry Ford did not invent the automobile. Ford's contribution to automobile production was the assembly line with interchangeable parts, the clear segmentation of labor and parts assembly, and eventually the flexibility of the assembly line to produce different variations of the automobile. Prior to the automobile assembly line, car manufacturers hand-crafted each engine, frame and chassis, and interchangeability was serendipitous. The state of applied cryptography today sits somewhere between hand-craftmanship and narrow standardizations of algorithms for DAR and DIT, respectively: applied cryptography does not have an assembly line for processing data with a series of cryptographic primitives in a consistent and interchangeable way for both DAR and DIT. SDFT presents the equivalent of an assembly line for applying cryptographic primitives to application data objects in a portable, independent, flexible, and unifying manner.

2.2. Parametric Permutations, Dirty Functions, and Data Scoping

Applied cryptographic primitives, especially ciphers, are 'dirty' functions requiring a complex set of parameters to control its behavior and its output. One might consider such particular and peculiar set of complex parameters required as a sort of 'baggage' for each combination of cryptographic function call and output message produced from it. For example, AES, the standard symmetric cipher, has 3 popular key sizes (128, 192, or 256 bits), 10 modes of operation, and a host of other controlling parameters such as IV, block character, block size, etc. [19] [20] [21] [22] These parameters for AES must be stored and set somewhere for each application implementing AES cryptography. The enormous number of permutations of the parameter set make it exceedingly tricky to set an industry standard of parametric settings [18]. Due to this 'baggage', AES implementations are commonly plagued by incompatibilities between vendor systems.

SDFT's approach allows for all parameter variants of AES (or any other cipher offered in the SDFT library) providing flexibility and compatibility. To make any two SDFT messages compatible with one another SDFT proposes a set of well-defined commands called transmutation commands, or transmutations (TMX) [13], which are sequenced and embedded alongside the output message by 'folding' the two data pieces together. The TMX command sequence creates a Transmutation Audit Record (TAR) to precisely specify the AES symmetric cipher transmutation and selected parameters to create an encrypted output message. TARs do not contain the encrypting/decrypting key parameter lest it compromises the encrypted message itself. The output message contains other message related parameters such as block size, blocking character and IV, what SDFT calls *data scoping* [13].

Data scoping, or *variable scoping for datasets*, is a systematic method of embedding message-level parameters directly within the output message. Data scoping is similar to how modern programming languages allow functional variable scoping, but it has some significant implications for application programmers: their applications **no longer need** to keep track of message level parameters related to ciphers thereby significantly reducing the onus on the developer and the application. Another implication is that every SDFT message may embed its own TAR—enabling one half of a dynamic, secure protocol

without prior negotiations is almost in hand at this point, which allows the receiving side to not know a priori exactly how the message was processed.

2.3. The Reversibility of Transmutation Commands and Dynamic Protocols

Transmutation commands (TMX commands) are characterized in part by their reversible properties. A symmetric cipher such as AES-256 may perform two operations: encrypt and decrypt. One is the reverse of the other assuming the right key and parameters are provided. SDFT defines two modes of operation, Figure 1: forward and reverse. These mode terms were chosen broadly to apply to more than symmetric ciphers so that TMX commands can be defined for data *transformation* functions as well as *cryptographic* functions.

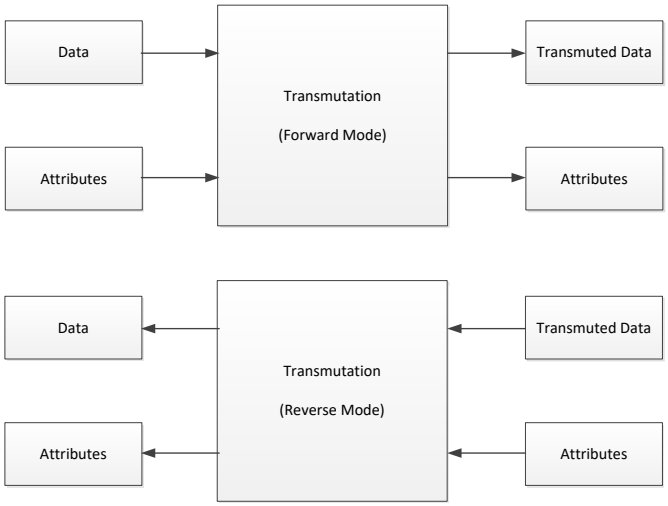

Figure 1 The two modes of a transmutation command

The reverse of AES-256 encrypt is an AES-256 decrypt. This logic can be applied systematically to most cryptographic primitives except for one-way functions, such as hashes and digital signatures, in which case we define their reverse modes as a verification operation. Rather than describe the nuances of how to analyze different cryptographic primitives to be normalized as a TMX command, we can explore what a TAR represents. A TAR containing a TMX command indicating AES-256 is sufficient to create an SDFT message that is encrypted with AES-256 and folded along with its encrypted, data-scoped, message. The recipient first unfolds the SDFT message at least once to separate the TAR from its output message, and then processes the TAR ***in reverse*** on the output message (while also providing the correct decrypting key) to produce the original cleartext message.

Sequences of TMX commands are called Transmutation Audit Records (TAR, not to be confused with the Unix tape archive ‘tar' command). TARs can operate forwardly, whereby each TMX command in a sequence operates in its forward mode, called a *forward traversal* of a TAR. Consequently, TARs can operate in a reverse manner whereby the sequence is reversed and each TMX command operates in its reverse mode called a *reverse traversal* of a TAR. Thereby each message has the ability to fully fold (ravel) and unfold (unravel) itself.

A TAR, Fig. 2, operates on a Single Data Structure called NSstr, and within it is at least one object to operate on. The structure's object serves as both input and output (similar to the car chassis moving on an assembly line) for each of the TMX commands in the TAR. For each TMX command that operates on the object, the object is first input to the TMX command, and then the same object is replaced

by the output of the TMX command which just processed the object. Thereby, the single data structure's object is continually transmuted by each TMX command during the forward traversal of the TAR.

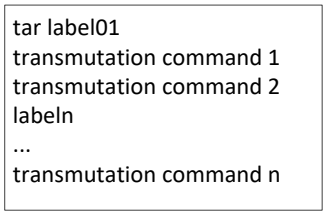

Figure 2: Structure of a Transmutation Audit Record (TAR)

Each TMX command can have input attributes besides the input object and can produce output attributes alongside the output object. These attributes can vary from encoding types to cryptographic keys to salt values; for example, in AES there are key sizes, mode of operation, and a host of other controlling parameters such as IV, block character, block size, etc. Incidentally, the output attributes in Figure 1 are the very same input attributes for an AES TMX, developers are not typically trained to view it this way.

TARs make it possible to determine which TMX commands require cryptographic keys, how many of them, what type, and in which sequence they are needed. Moreover, TARs give us a basis to provide two convenience features: key stack form validation and the automatic generation of cryptographic keys, if any are missing. We will delve into API key management in a later section, it is different from the conventional KMS key management: SDFT API key management is the management of cryptographic keys within a complex cryptographic function call.

Lastly, the NSstr structure also contains the TAR, which produced the included object.

Reversible computing means different things to different people. It is a recognized phrase referring to the largest set of concepts related to computing in which backward execution is used [23]. A restricted form of Reversible Computing is constrained to work on a single data structure and where state transitions can be sequenced and fully reproduced [24]. The reversible characteristic of a TMX command allows a TAR to be reversible as well. This results in a message folded by a TAR and further unfolded by the same TAR in a reverse traversal, Fig. 3. Full reversibility of a TAR can only be maintained as long as there is no data loss during TMX processing in the object and its attributes.

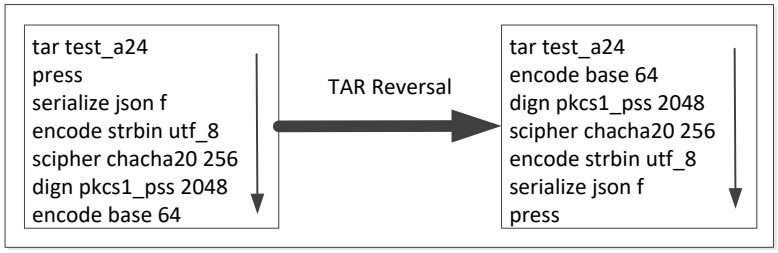

Figure 3: TAR reversal illustration

A library of normalized, reversible TMX commands covering serialization, encodings, compressions (lossless), symmetric ciphers, asymmetric ciphers, digital signatures, locks, and digests (hashes and keyed hashes) can provide the raw ingredients to allow almost any combination of operations to produce a bitstream message that is protected or unprotected. Formed into TARs, particularly useful and well-thought-out sequences may be saved, shared, and injected into any NSstr structure for message

processing. Since the SDFT message eventually encapsulates the TAR, which created it along with the processed message, the recipient need not know a priori the precise sequence of operations needed to recover the encoded message.

A TAR is a narrow implementation of ***reversible computing*** [25] confined to data transformation functions within a limited memory space operating on a single piece of data. Developers composing a TAR need only construct the forward mode and the reverse is implied and automatic. This reduces the work of creating a message encryption/decryption process by **50%**. Anyone experienced in constructing both sides of a cryptographic message system has experienced finding bugs in creating mirror operations on the writer/reader functions. TAR reversibility allows for an automatic dynamic construction of the reader (reverse mode) based on the construction of the writer (forward mode), a significant technological advancement in message protocols.

TARs are text-based command sequences, so they can be dynamically changed without custom programming at runtime; a change of cryptography (transmutation command) no longer requires re-compiling code and can be introduced in real-time saving much work and time. TARs can be changed on a per SDFT message basis leading to a true dynamic, secure, negotiation-less protocol. Sensitive messages can apply more secure TARs at will, such as using multiple AES-256 commands in sequence with separate keys for each call, or with a future symmetric-512 cipher. If the SDFT library is sufficiently updated with the latest transmutation commands (i.e. cipher variants), the readers of SDFT messages do not need to know the method of message creation a priori.

Since TARs can be inspected prior to processing, any system using SDFT messages can automatically negotiate on which TARs to use thus generalizing the solution of negotiated protocols in a universal way. A set of pre-defined, pre-selected and approved TARs can be shared between communication partners to expedite such negotiations automatically. Systems employing dynamic protocols are more compatible with one another with each operating at their preferred TAR choices or one party insisting on a desired TAR. Examining a TAR before processing can look for errant or illegal TAR constructs to prevent abnormal behavior, enhance security and resiliency.

2.4. Data Transformations

Data transformation functions are applied to various application in-memory objects to prepare the data for cryptographic processing. Until now, no library offers data transformation functions in a conveniently organized manner that is also normalized alongside cryptographic functions to work seamlessly as transmutation commands. Our approach creates the equivalent of an ‘assembly line’ for generating encrypted messages from application objects, greatly simplifying the process of applying cryptography with all the inherent benefits of an assembly line.

SDFT includes a normalized set of transmutations for such functions as serializers, encoders and compressions. Thus, a TAR has the capability of describing how to process an in-memory object into an encrypted message folded along with the TAR that created it from start to finish with a reversibility built-in so that writing an SDFT message means it can be read just as easily. The use of lossy compressions can adversely affect the reversibility of a TAR but lossy compressions can be treated as a special type of one-way function with a null operation in reverse mode.

The components and layers of SDFT are straightforward to describe, the approach is logical, and each technique is well-known and practiced, however, once put together in the novel way described, some unexpected ancillary characteristics reveal themselves, leading to surprising and useful features.

2.5. Key Management in the API

Applying cryptography usually requires the management of cryptographic keys. Key management, sometimes called public key infrastructure (PKI), requires at a minimum secure key storage, key generation, key identification, and key distribution. Most publicly available programmatic cryptographic functions do not automatically generate the keys necessary to operate the ciphering algorithm it performs. The functions are generally written to accept properly formed keys as parameters.

Key generation is a non-trivial task for a programmer; the right methods and proper key specifications must be applied in the generation of keys otherwise the full security of the cipher will not be realized. Further, key management in the scope of feeding a sequence of keys for a sequence of cryptographic operations is not a simple task and must be done with care. SDFT innovations in API oriented key generation and management streamlines this process for all programmers.

For example, if a developer calls a cipher, say AES-128, to encrypt a piece of data with the appropriate parameters without specifying the encrypting key, why does the function not generate a proper key, perform the operation, and return the encrypted data along with the newly generated key? It is a simple concept but generally unavailable in common cryptographic libraries thus resulting in further burdens on the programmer. In SDFT, if an AES transmutation is operated on without any input key, then a proper one is generated and used in the operation. For a TAR that uses six TMX commands that each require a key, six properly formed keys are generated, used and returned in a proper sequence along with the output message.

In SDFT, key generation, key structure validation, key sequencing for functions that perform multiple cryptographic operations on an input data, are all automated and made convenient for the developer. The SDFT key management features make it convenient and easy to make use of many keys and cryptographic operations without getting bogged down in keeping track of the minute but important details.

To further simplify its applicability for developers, SDFT proposes standardized cryptographic key structures which act as keys or keyholes: the key structure without a key value is marked as a keyhole and vice versa, therefore ‘inserting’ a key into a keyhole is simply placing a key value and marking as such. The SDFT key structure automatically stamps an identifier for each key generated; another simple yet seldom performed feature by any key creation functions in cryptographic libraries.

Why is this important? The application of cryptography requires many simple steps in a specific sequence on a piece of data. The accumulation of many simple steps tends to burden the programmer and may result in steering the programmer to take shortcuts, or make mistakes leading to unsecure data handling, or make bad application design choices. SDFT, using TARs, compactly combines simple steps that are necessary to accomplish a series of complex operations, making it easy for the programmer.

We have yet to find any widely available library with the ability to process multiple cryptographic functions requiring multiple keys in a precise sequence all performed on a single input data. Any such instances of complex series of cryptographic operations are usually customized programs requiring a high level of skill and experience. The API Key management built into SDFT helps reduce developer errors in applying cryptographic operations properly and consistently with less knowledge and experience required.

2.6. Structured Cryptography

The initial SDFT US patent US10503933 [13] fully specifies innovations including sequencing of TMX commands, reversibility of TARs, data scoping, encapsulation of TARs with the output, variable locks, key management in the API, and automatic key derivations; these and other advanced innovations in SDFT lower the knowledge level required of application developers to code more securely and keep data safe.

The normalization of data transformation functions and cryptographic functions allows the formation of TMX commands. These commands are compact and complete enough to fully describe the conversion of an arbitrary data structure in an application memory to a transmittable or storable format. Further, the SDFT internal structures allow for the assembly line-like processing of Transmutation Audit Records (TAR) in both forward and reverse modes.

Efforts such as JSON Web Key (JWK) [14] structures specify the particular cipher applied to a payload but are limited to a unitary cipher operation. Extensive examinations and arguments by the USPTO are available in the patent jacket documenting the innovative nature of SDFT and its significant differences from prior art and research.

The use of SDFT primitives allows for the systematic application of well-known cryptography programming techniques in a repeatable fashion by any developer. As such, TMX commands form the building blocks of more sophisticated applications of cryptography on data due to its abstractions and handling of all the details. Interpreted languages such as Perl and Python, which were designed to automate memory management and be more forgiving in type checking, have led to enormous productivity for an entire generation of application and script programmers. Similarly, by providing automation and structure to a field where in most cases it is an art form chiseled by craftsmen, SDFT shows how to build a framework around applied cryptography to address the most common issues by ordinary developers using ***structured cryptography***.

For example, key encapsulations such as encrypting the encrypting symmetric key with a public-private key, are ubiquitous in applied cryptography such as in PGP. Using a structured cryptography approach, the key encapsulation can be expressed as a SDFT call using a TAR with an asymmetric cipher transmutation on a SDFT key data structure. PGP cannot easily accommodate new ciphers without serious ramifications to all PGP-compliant applications and pre-existing PGP encrypted data whereas SDFT's structured cryptography presents a generalized key encapsulation solution capable of injecting PQC transmutations in a seamless way to provide security from quantum computing in the future.

Figure 4 illustrates a series of tasks to perform on a Python data object 'data' to be transformed into a string that can be written onto disk. This example uses mostly data transformation functions and one simple CRC16 hashing function on the dictionary variable 'data' (Application Data Object). The 'Normal Python code' shows a straightforward sequence of python commands for a writing section and a reading section comprising 18 lines. The 'SDFT code using TAR' sections show the TAR commands that perform the equivalent tasks and the associated SDFT calls using the TAR definition 'test_a70'.

Make data JSON compatible by converting all bytes to strings
Perform JSON serialization on data
Convert JSON string into bytes string
Calculate CRC 16 hash on data
Wrap data and digest into a structure
Write data to file

Tasks to be performed on Data Object to write a message to disk

```
data = dict(string = 'oops', bytes = b'234', deeper = ['str2', b'2'])
```

Application Data Object

```
### Perform manual data folding (ravel)
import base64, json, binascii
data['bytes'] = base64.b64encode(data['bytes']).decode()
data['deeper'][1] = base64.b64encode(data['deeper'][1]).decode()
data = json.dumps(data, sort_keys=False)
data = data.encode('utf_8')
digest = binascii.crc_hqx(data,0).to_bytes(2, byteorder = 'big')
wrap = dict(data=data.decode(), digest=base64.b64encode(digest).decode())
with open("mydata.json", "w") as f:
   json.dump(wrap, f)

### Perform manual data unfolding
import base64, json
with open("mydata.json", "r") as f:
   wrap = json.load(f)
digest = binascii.crc_hqx(wrap['data'].encode(),0).to_bytes(2, byteorder = 'big')
if digest != base64.b64decode(wrap['digest']):
            print('error: crc codes do not match')
data = json.loads(wrap['data'])
data['deeper'][1] = base64.b64decode(data['deeper'][1])
data['bytes'] = base64.b64decode(data['bytes'])
```

Normal Python code

```
tar test_a70
press
serialize json f
encode strbin utf_8
digest hash crc 16
encode base 64
```

TAR commands

```
### Perform ravel via SDFT (data folding)
import NSsdf
ns = NSsdf.NSstring(data)
retobj = ns.ravel(tarName="test_a70")
ns.writeJSONfile("mydata.json")

### Perform unravel via SDFT (data unfolding)
import NSutil, NSsdf
ns = NSsdf.NSstring(NSutil.readJSONfile("mydata.json"))
retobj = ns.unravel()
data = ns.getObj()
```

SDFT code using TAR

Figure 4

A quick comparison shows that the SDFT python code is only 8 lines, a reduction of 56% in written code. Note that the ‘SDFT code using TAR’ will work for any valid TAR thereby reusability of code is instantaneous and consistent. The TAR ‘test_a70’ is a human friendly, readable command sequence, and the TAR is written in one direction, forward, for creating or folding the data. The unfolding TAR is created automatically and dynamically from the folding TAR thereby reducing the programmer’s workload by at least 50%.

Symmetric ciphers, asymmetric ciphers and any other cryptographic primitives are available as TMX commands in Figure 5 along with the current choices of operations. ‘scipher’ performs any of the three available symmetric ciphers: aes, chacha20, or salsa20. Also in Figure 5 are two sample TARs showing symmetric cipher and digital signature TMX commands usage. The SDFT code from Figure 1 can operate on these TARs ‘test_a20’ or ‘test_a24’ without modifying the Python code whereas significant changes are needed in the ‘Normal Python code’.

Transmutation	Operations
serialize	JSON, XML, COM, CORBA, SOAP
compress	zlib, gzip, bz2, lzma
encode	base64, base85, utf_8, quopri, binary, over 90 variations of codecs
acipher	pkcs1_oaep, pkcs1_v1_5
scipher	aes, chacha20, salsa20
derive	pbkdf2, hkdf, scrypt
digest	hash: crc, md5, sha1, sha2, shae3, shake128, shake256, keccak hmac: md5, sha1, sha2 cmac: aes
dign	pkcs1_v1_5, pkcs1_pss, dss,

```
tar test_a20
press
serialize json f
encode strbin utf_8
scipher chacha20 256
encode base 64


tar test_a24
press
serialize json f
encode strbin utf_8
scipher chacha20 256
scipher aes 256 mode=eax
compress zlib
dign pkcs1_pss 2048
encode base 64
```

Figure 5 Transmutation commands, operations, and sample TARs

Figure 6 illustrates the processing cycle of SDFT data folding and unfoldings in an application. The ‘retrieve’ and ‘store’ operations can be interpreted as read/write and receive/send types of operations for DAR and DIT respectively.

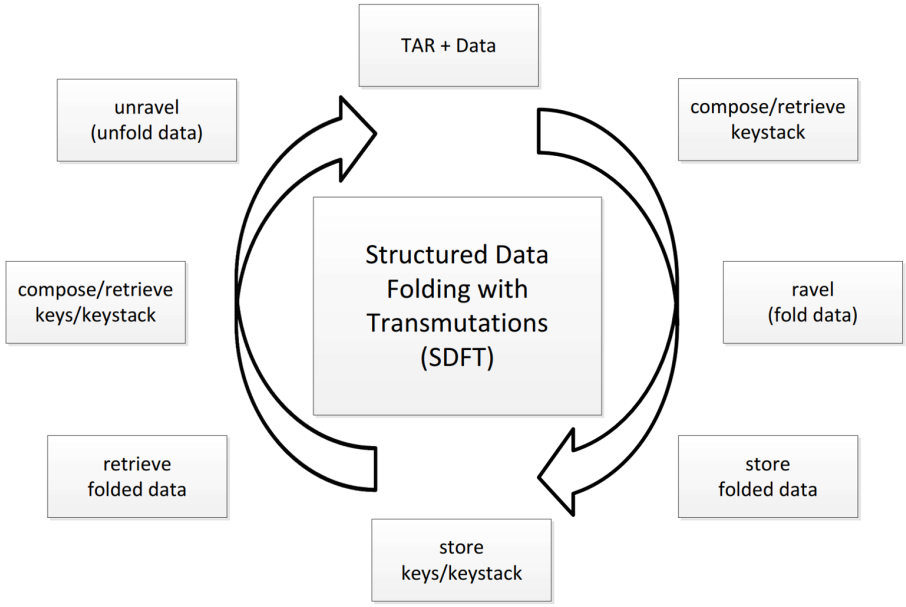

Figure 6 The SDFT processing cycle of data folding and unfolding

2.7. Backward & Forward: Time Compatibility of Data & Transitions in Cryptography

Since SDFT views messages as data that can be in-transit or at-rest with the only variable being the time *t* of reading, there are three significant implications of SDFT messages saved onto persistent stores for a length of time.

First, SDFT messages with TARs prescribing older, defunct ciphers can still be read by the current SDFT library and properly processed given the correct cryptographic keys. Application programming is littered with obsolete file formats and even more so when it involves encrypted file formats: there is no guarantee that a current application can always read an obsolete file format. Maintaining older applications for backwards compatibility is expensive, time consuming and inefficient. SDFT isolates the backwards compatibility at the message level relieving the application of such responsibilities; therefore, a current SDFT library automatically gives read ability for old and current SDFT messages.

Second, storing encrypted data via SDFT messages allows for continuous transitions to new cryptography indefinitely or forward compatibility. Reading older SDFT messages and writing them back out allows SDFT to replace the older TAR with the latest TAR with updated ciphers to transition it to the latest cryptography automatically. The only maintenance is the proper updates and upgrades to the SDFT library to integrate any new cryptography as transmutation commands. This feature can allow the computing industry to keep up with current cryptography standards on their secured data on a per message (file or document or transient messages) level thereby providing the flexibility to implement new cryptography standards in a systematic, prioritized, and/or incremental way. **Institutions can integrate SDFT today using RSA and AES transmutations and be prepared for a seamless transition to PQC transmutation commands when needed in the near future on their schedule.**

Third, many systems require encrypted data in persistent stores to undergo massive one-time transitions in an expensive and interruptive fashion. Massive transitions are prone to be time consuming, disruptive, expensive, and error prone. SDFT innovates by allowing its dynamic forward compatibility feature to drive an incremental process to transition encrypted data in persistent stores in a systematic, incremental, continuous, and flexible process.

SDFT's backwards and forward compatibility, or time compatibility, adds a powerful ***solution that can anticipate future cryptography transitions indefinitely*** and one which is well-aligned with the **NCCoE Migration to PQC** effort.

2.8. Integrating PQC DSA into the SDFT Python Library

Utilizing the existing SDFT Python module as a starting point, the project analyzed the candidate PQC algorithms from Round 3 and applied the TOP normalization process for new ciphers to modularize the PQC algorithms into TMX commands.

The PQC DSA candidates (and eventual standards) presented functional I/O similar to existing 'dign' transmutations (digital signature) Figures 7 & 8.

Transmutation	**operation=**	**keylen=**	**hashtyp=**	**digestlen=**	**saltlen=**	keytyp
dign	pkcs1_v1_5 **pkcs1_pss**	[1024| **2048**|3072]	any	[160-**512**]	- **digestLen**	RSA

Figure 7 RSA based digital signature algorithms (dign TMX)

Transmutation	**operation=**	**keylen=**	**hashtyp=**	**digestlen=**	**mode=**	keytyp
dign	dss	[1024|**2048**|3072]	**sha1/2** (5)	[160-**512**]	[**fips-186-3**| deterministic-rfc6979]	DSA
		256 (curve-256)				ECC

Figure 8 DSA/ECC based digns

After mapping out the various parameters, we arrive at these new dign transmutation commands for the PQC algorithms, Figure 9,

transmutation	operation	keytyp	# key types	variant	hashtyp	trait	library
dign	ml_dsa	dilithium	3	44	any	-	**pqcrypto**
				65			
				87			**quantcrypt**
	fn_dsa	falcon	2	512	any	-	**pqcrypto**
				1024			[**quantcrypt** | pqcrypto]
	slh_dsa	sphincsp	3	128	sha2	fast	**pqcrypto**
						small	
					shake	fast	
						small	
				192	sha2	fast	
						small	
					shake	fast	
						small	
				256	**sha2**	**fast**	
						small	
					shake	fast	[quantcrypt | **pqcrypto**]
						small	

Figure 9 PQC DSA normalized as TMX dign

In August, the NIST published CRYSTALS-DILITHIUM and SPHINCS+ as FIPS-204 (ML-DSA) [26] and FIPS-205 (SLH-DSA) [27] PQC DSA standards respectively. Falcon is slated to be standardized next.

Note that the TMX command is a dign and the operation type varies. ‘keytyp’ indicates the type of key that the particular ‘operation’ requires. Listed are also variant options suggested by NIST for varying security levels. ‘trait’ is a parameter to specify a further variation of the PQC algorithm. Finally, there is a choice of two Python modules offering test implementations of the PQC DSA algorithms. We did eventually added a third Python module called ‘liboqs’ [28] [29] described later in this report.

These 16 PQC DSA algorithms (variants) are the released standards and one candidate algorithm so far to replace the ECC and RSA based DSAs. There are many choices providing varying performance and resource requirements for each algorithm. More information on each can be found in the FIPS documents and candidate algorithms write-ups on the NIST PQC website [30].

Accommodating the PQC DSA fit very well into the current dign TMX I/O parameters and it was relatively straightforward to integrate them into the SDFT library. Tests were created to automatically run including key generation, signature, and authentication unit tests on various data sets, each library and every variant. A typical run result in Figure 10,

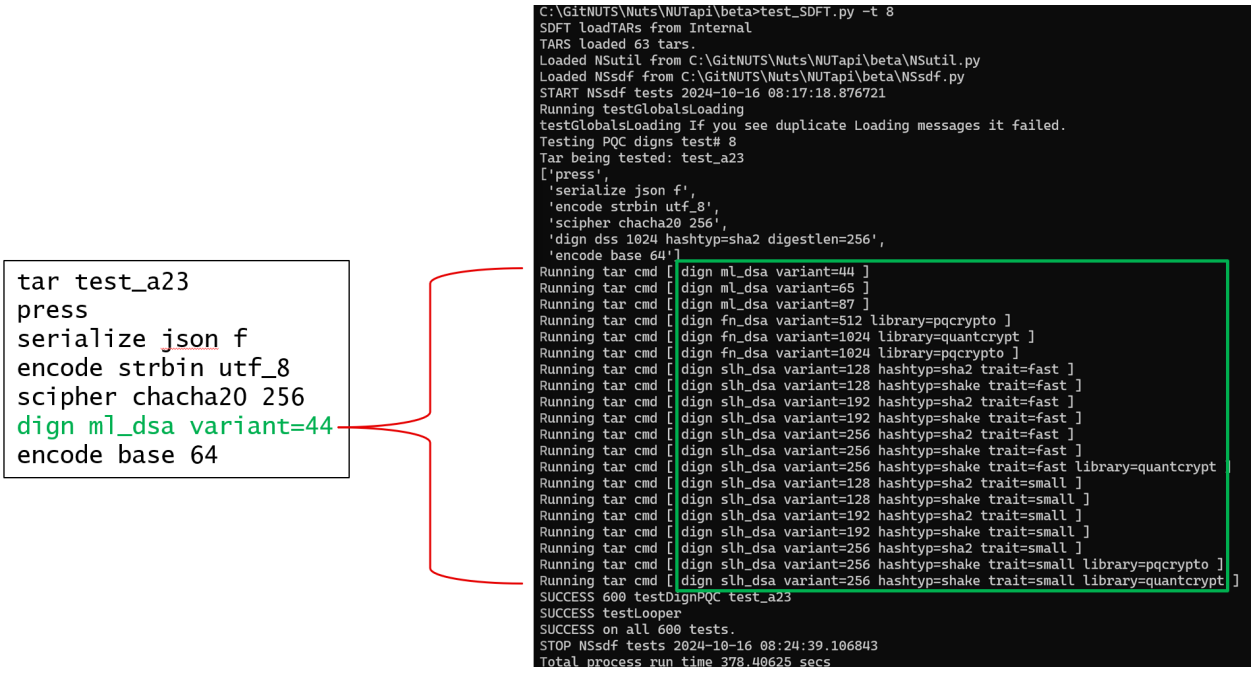

Figure 10 PQC DSA test run log using test TAR.

In Figure 10, the TAR named 'test_a23' serializes an input data structure into JSON, encodes it into a binary string, perform a ChaCha20 symmetric encryption on the binary string, creates a digital signature using a PQC DSA variant algorithm, and then encodes the resulting output data structure into base64. The output of an SDFT 'ravel' (fold) call is an NSstr data structure where a copy of the TAR is folded alongside the digital signature, and the encrypted data string. The NSstr structure is serializable as JSON for storage on a disk or transmission across the wire.

In effect, the SDFT NSstr message is a self-describing, self-prescribing message due to having folded the TAR instructions which created it. Obviously, no keys are folded within it since that would be defeating the purpose of securing the message. A stack of keys is returned as part of the ravel call which in this case will include a 256 bit symmetric key for ChaCha20 and then a ML-DSA-44 PQC DSA key. Whether or not these keys are generated depends on whether the initial 'ravel' call was done with an input key stack containing either one of the keys or both or none. The SDFT ravel call is designed to assess whether any of the required keys need to be generated, whether keys provided have the right form and part (in regards to asymmetric keys, the call determines whether the correct key part has been presented), and to sequence the keys within the key stack in the order they are needed by the TAR sequence.

In the testing, TAR test_a23 is an array of text strings where only the 6th line, "dign ml_dsa variant=44", is replaced by the PQC DSA variant desired within the testing loop (indicated by the green outline). Therefore, the change of cryptography is dynamic and requires no coding at all within an application but rather by changing a text-based TAR sequence which is eventually folded with the message produced.

Each test runs through an 'unravel' call which reverses the process of the ravel call. In the case of a TAR test_a23: decode the base64 strings in NSstr into binary strings, perform a MS-DSA-44 PQC DSA Verification on the digital signature string within and against the encrypted message, upon successful verification perform a decryption using ChaCha20 on the encrypted message, convert the binary message into an utf_8 string, deserialize the message from JSON into a Python structure (array, variable, string,

dictionary, etc.), convert encoded 'bytes' strings into 'bytes' Python strings (press transmutation in Python), and then return an NSstr data structure.

As easily seen, this sequence of processing of an unravel call is basically following the TAR test_a23 backwards, performing the defined reverse operation of each TMX command producing the original data in the structure it was provided to in the original ravel call. Obviously, the unravel call expects a proper key stack populated with the exact keys needed in the correct forward TAR sequence. In the case of a dign transmutation like ML-DSA-44, the call expects to have the private part of the asymmetric key set when doing a forward traversal of the TAR. Any verifiable failures will result in raising an error and the termination of the unravel call. For this particular case, it can happen during the dign verify. Since ChaCha20 does not have a built-in data integrity feature, the dign will have to provide that capability during the TAR processing; hence the compactness of the TAR allows the application of logical cryptographic operations in a sequence that is commonly used.

All parameters and attributes necessary for executing the ravel/unravel process are embedded within the NSstr data structure using data scoping. This approach abstracts away cumbersome cipher parameter management from the application code, keeping it clean and straightforward. Moreover, because all cryptographic parameters are folded into the message, any receiving or reading program can process the message correctly—provided it can supply the necessary keys for the unravel call.

2.9.Integrating PQC KEM (Key Encapsulation Mechanism)

Following the NIST guidelines in the FIPS-203 standard ([31] pg. 15 section 3.3) for CRYSTALS-KYBER, the K-PKE component is not approved for independent use as a public key encryption scheme. The ML-KEM standard has three algorithms: ML-KEM.KeyGen(), ML-KEM.Encaps, and ML-KEM.Decaps. Figure 11 illustrates the workflow for these algorithms to establish a shared secret (a 256 bit symmetric key) between two participants, Alice and Bob:

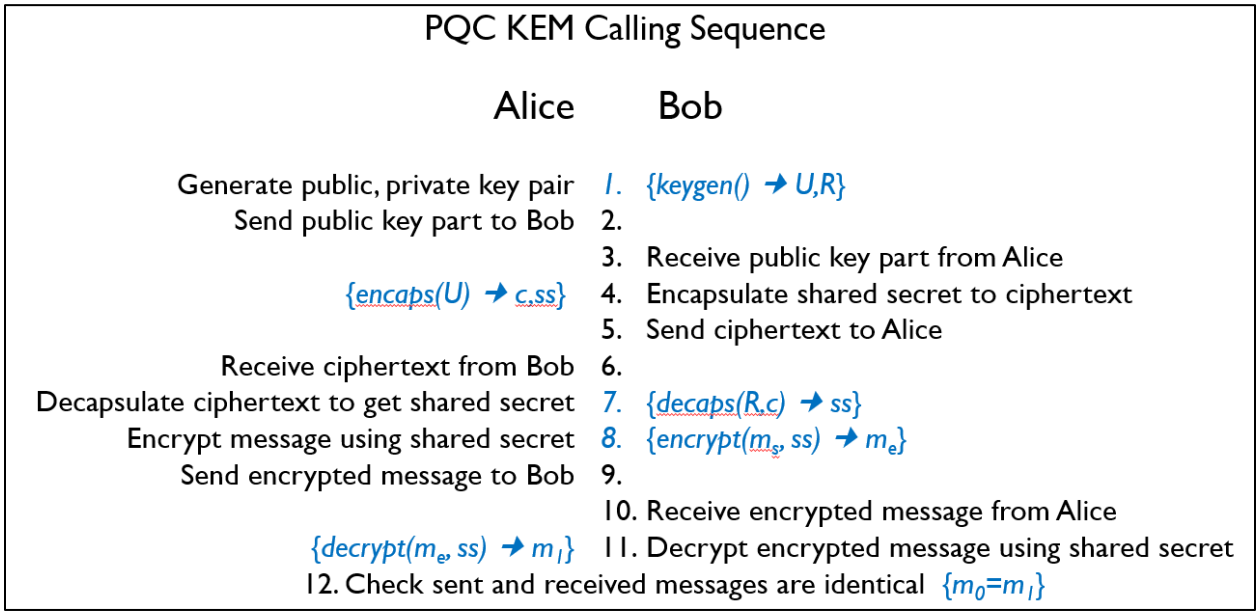

Figure 11 Calling sequence for PQC KEM algorithm. Also the test in the SDFT KEM runs.

The KEM forces a specific sequence to establish the shared secret, ***ss***, between Alice and Bob. By utilizing the three algorithms of ML-KEM in the sample sequence above, Alice and Bob will be able to communicate using some symmetric cipher taking a 256 bit key, the shared secret ***ss***.

The acipher transmutation (for asymmetric cipher, Figure 12) operations such as RSA operated closer to the K-PKE component level algorithm and was thus characterized to be used as a public key encryption scheme where one can directly encrypt a symmetric key and then decrypt it.

Transmutation	**operation=**	**keylen=**	**hashtyp=**	**digestlen=**	**e=**	keytyp
acipher	**pkcs1_oaep**	[1024|**2048**|3072]	**sha1/2**/3 (9)	[160|224|256|384|**512**]	**65537**	RSA
	pkcs1_v1_5		-	-	-	

Figure 12 Asymmetric cipher TMX, acipher, for RSA

Since the FIPS-203 standard only allows using the KEM level algorithms KeyGen, Encaps and Decaps, our TOP analysis pointed to requiring a new transmutation we will call ***akem*** for asymmetric KEM. We normalized Kyber and 3 candidate KEMs as shown in Figure 13:

libkem: Library Options for PQC KEM Algorithms

transmutation	operation	keytyp	# key types	variant	trait	library
akem	ml_kem	kyber	3	512	-	[pyqc | **pqcrypto**]
				768		
				1024		[quantcrypt | pyqc | **pqcrypto**]
	bike_kem	bike	3	3114	-	N/A
				6198		
				10276		
	cmc_kem	classicmc	5	**348864**	**null**	**pqcrypto**
					f	
				460896	null	
					f	
				6688128	null	
					f	
				6960119	null	
					f	
				8192128	null	
					f	
	hqc_kem	hqc	3	128	-	N/A
				192		
				256		

Figure 13 Asymmetric KEM TMX, akem, for PQC algorithms normalized.

Initially we did not find any available Python PQC implementation for ***bike*** and ***hqc***, but later we did find one for **hqc** in the **liboqs** module.

Figure 13 shows that the replacement for RSA/ECC public key encryption schemes can come in 4 varieties with 19 variants. Quite a choice to make. Some of the key sizes for Classic-McEliece can grow to well over 1Mb in size and there are performance considerations for processing such algorithms. Within the SDFT library, those considerations are left up to the user/application to constrain if necessary. SDFT allows any size keys using its Key Interchange Specification Structure (KISS) [13] which can act as both a key and a keyhole. In fact, during this PQC integration into SDFT, the only changes made to the key storage and manipulation of KISS constructs was to completely separate the operational relationship between private and public keys when using PQC. More specifically, in RSA, it is common to just keep the private key part because the public key part can be quickly derived from the private key part. In PQC, there is no equivalent general rule of that nature concerning the relationship between public and private key parts so none was made nor assumed. In terms of key storage and key manipulations, the existing SDFT code required no modifications to accommodate PQC keys.

SDFT akem Call Steps and Processes

	Step	Mode	Operation	Input		Output		Description
				keystack	obj	keystack	obj	
Alice	TX_1	forward	keygen	none	none	U,R	$ns(U_{KISS})$	Create key pair, send U to RX
Bob	RX_2	reverse	encaps	$blank_{KISS}$	$ns(U_{KISS})$	U	U_{KISS}	Convert received U into kiss key
Bob	RX_3	forward	encaps	U	none	U,S,C	$ns(C_{KISS})$	Create secret with U, send C to TX
Alice	TX_4	reverse	decaps	U,R	$ns(C_{KISS})$	U,R,S,C	U,R,S,C	Recover S from received C
Nut Access Control	TX_5	forward	decaps	U,R	C	U,R,S,C	$ns(URSC_{KISS})$	Recover S from given C
	TX_6	forward	encaps	U,R	none	U,R,S,C	$ns(C_{KISS})$	Create secret with U

TX is transmitter
RX is receiver
Processing is determined by conditions.
Transmittable SDFT message: **ns(message)**

Figure 14 A TMX normalized calling sequence for an akem TMX.

Following the TOP methodology, we transformed the Alice-Bob KEM sequence into a normalized transmutation form shown in Figure 14. The notation in use: U, public key part; R, private key part; S, shared secret; C, ciphertext (encapsulated encrypted S); TX, transmit; RX, receive; ns(), NSstr structure; KISS, Key Interchange Specification Structure.

The newly formed ***akem*** transmutation can now be used to implement the above call steps and processes and then use the shared secret in a message encryption call to test all the variants of PQC KEM algorithms we found in Python modules in Figure 15.

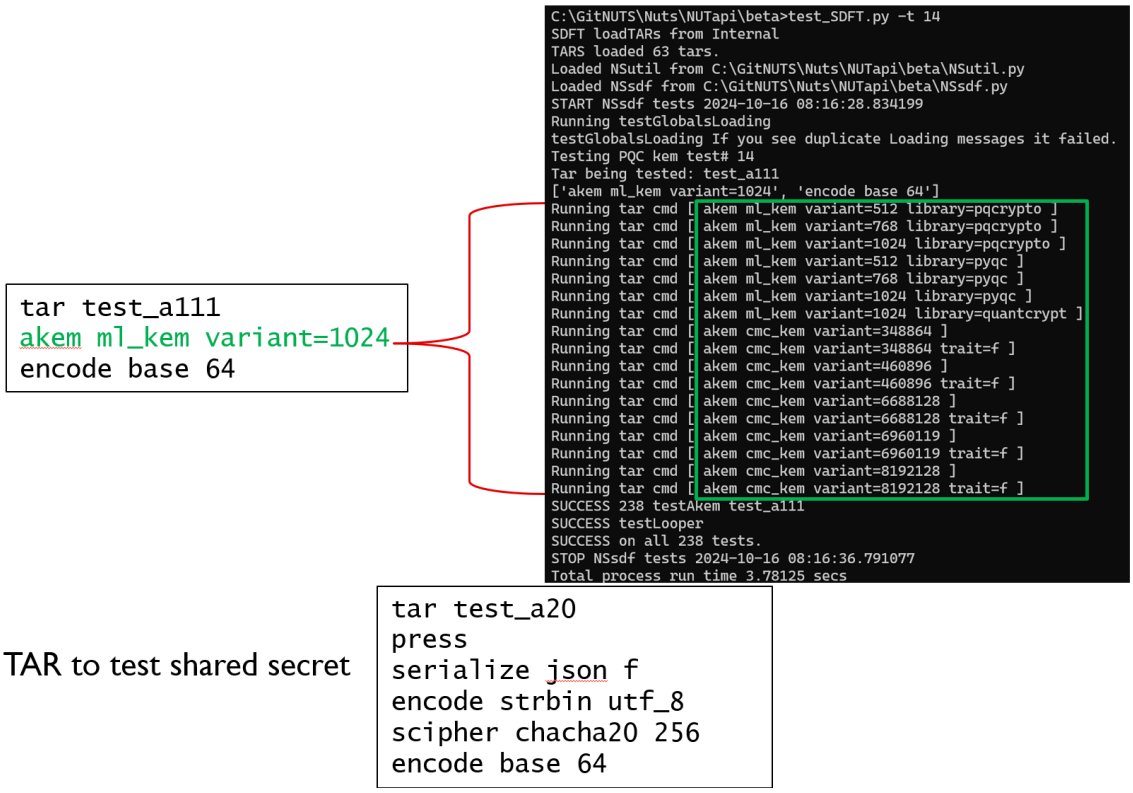

Figure 15 PQC KEM test run log using test TAR.

For steps 1-4 (TX,RX,RX,TX) in Figure 14, the test uses TAR ‘test_a111’ only, Figure 15. The ‘mode’ and the ‘input’ determine the operation to perform using the akem transmutation. Once the shared secret S is established, TAR test_a20 is used with S to encrypt and then decrypt an arbitrary message. In this particular case, verification is done by comparing the original message vs. the decrypted message rather than using a digest or dign transmutation.

Overall, the successful integration of the PQC KEMs and DSAs into SDFT shows that the application of new cryptography can be made modular, dynamic, compact, and portable. Any SDFT message can change its cipher, any SDFT stored message can be read and rewritten with updated ciphers. An updated list of transmutation commands after integrating PQC algoritms are presentd in Figure 16.

Transmutation	Operations
serialize	JSON, XML, COM, CORBA, SOAP
compress	zlib, gzip, bz2, lzma
encode	base64, base85, utf_8, quopri, binary, over 90 variations of codecs
acipher	RSA: pkcs1_oaep, pkcs1_v1_5
scipher	aes, chacha20, salsa20
akem	PQC: ml_kem, bike_kem, cmc_kem, hqc_kem
derive	pbkdf2, hkdf, scrypt
digest	hash: crc, md5, sha1, sha2, shae3, shake128, shake256, keccak hmac: md5, sha1, sha2 cmac: aes
dign	RSA: pkcs1_v1_5, pkcs1_pss DSA/ECC: dss PQC: ml_dsa, slh_dsa, fn_dsa

Figure 16 TMX amended to include new PQC TMX.

2.10. A test of integration: liboqs

As a test to gauge SDFT to integrate any new PQC Python module or library, we found a module called 'liboqs' built upon from Open Quantum Safe (liboqs | Open Quantum Safe) C library and ported to Python as a liboqs Python module [28]. The liboqs module was a nice package where it included most of the PQC algorighms we integrated already as shown in Figure 17.

				SDFT parameter "library="			
Function	Algorithm		Variant	quantcrypt	pyqc	liboqs	pqcrypto
KEM	kyber		512		x	x	x
			768		x	x	x
			1024	x	x	x	x
	bike		3114				
			6198				
	classic-mceliece	normal	348864			x	x
			460896			x	x
			6688128			x	x
			6960119			x	x
			8192128			x	x
		fast	348864			x	x
			460896			x	x
			6688128			x	x
			6960119			x	x
			8192128			x	x
	hqc		128			*	
			192			*	
			256			*	
DSA	dilithium		44			x	x
			65			x	x
			87	x		x	
	falcon		512			x	x
			1024	x		x	x
	sphincs+	fast shake	128			x	x
			192			x	x
			256	x		x	x
		fast sha2	128			x	x
			192			x	x
			256			x	x
		small shake	128			x	x
			192			x	x
			256	x		x	x
		small sha2	128			x	x
			192			x	x
			256			x	x

Figure 17 TMX commands amended to include liboqs PQC algorithms.

The liboqs module also provided an implementation of the hqc KEM algorithm which produces 512 bit symmetric keys. For our testing, the 512 bit string was reduced to 256 bits to allow ease of use in the ChaCha20 cipher.

We began timing from the downloading of liboqs Python module, following the instructions for compiling it for Python installation which includes making the C library components, testing the Python module installation using their test code, figuring out the peculiarities of calling sequences for the liboqs calls, then integrating it into SDFT, adding to the SDFT PQC test cases, verifying the tests were running properly, and running the full suite of tests. This took 5 hours where about 2 hours were spent on non-SDFT integration tasks such as installing, compiling and figuring out the liboqs API. In all, it took less than 3 hours to integrate 33 PQC variant algorithms into SDFT and have it fully tested. We can conclude that the integration capability for new PQC algorithms into SDFT is efficient and cost effective.

Our experiment with integrating liboqs requires an examination of the API presented because we were surprised at the variability of API presentations even within the 4 Python modules we integrated. Some presented class libraries, others just required module imports. NIST recommended API forms were inconsistently implemented such as switching around the arguments in a call. In one module, binary strings were returned as an array of integers. Some objects (cipher class instantiations) required presetting

keys before a call, some did not. Therefore, after just working with a few libraries, even a well defined API guidance provided by NIST resulted in unexpected variations of interpretation. For PQC standards, there are very few parameters to choose for a given algorithm (unlike AES), but it still resulted in many forms and styles of implementation.

These characteristics of SDFT messages allow for setting up dynamic protocols where two communicating parties can change cryptography at will during the same session. Documents and data stored onto persistent media as SDFT folded messages can be systematically and incrementally updated in cryptography at next touch and/or on a prioritized basis without any system access down times. This is a significant advantage to have when considering large repositories of encrypted unstructured data. SDFT sets the foundations of providing a consistent method of processing data for both Data-At-Rest and Data-In-Transit, something that has been lacking in applied cryptography. The use of TARs provids a one step process for transforming an application data structure to a storable/transmissible message. By using well designed TARs, a larger group of developers can apply consistent security to their data without the requisite experience and knowledge requirements of conventional methods. SDFT TARs are written in one direction and the reverse is implied: this capability can reduce potential errors in coding by 50% to start.

2.11. Conclusion

Once SDFT is integrated into a system, that system gains the advantage of future proofing from any change of cryptography that can be normalized into the SDFT library using the TOP methodology. From an efficiency and cost perspective, SDFT can deliver significant advantages at scale indefinitely.

For these reasons, we believe that SDFT may present an opportunity to set a standard for applying cryptography at the message level in a universal way for both DAR and DIT.

3. Addendum

3.1. Applicability of SDFT: The Internet, Blockchain and Data-Centric Security

The applicability of SDFT to secure data communication and secure data storage is straightforward given the inherent benefits outlined in this report. This section presents a use case for how SDFT can help (1) blockchain, and followed by a brief introduction to the initial motivation for developing SDFT, (2) data-centric security.

3.2. Qryptosaurus: A Blockchain Extinction-Level Event

The last 15 years have witnessed a dramatic rise of digital assets in the form of cryptocurrencies and NFTs. In a cryptocurrency such as Bitcoin, it is the cryptography which maintains and secures the Bitcoin for the coin owner. The original paper for Bitcoin [32] does not specify a particular digital signature, however, the Bitcoin codebase [33] and wiki [34] specify Elliptic Curve Digital Signature Algorithm (ECDSA). In essence, a cryptocoin provides the most direct relationship between monetary value and the application of a cryptographic algorithm. With the cryptocurrency asset base at over 3 trillion USD and being signaled as a priority of the incoming US Administration, the securing of blockchain networks from CRQC will become increasingly critical to the blockchain industry.

During our work on the NIST grant to integrate PQC into the SDFT framework, it became evident that quantum computers will pose an Extinction-Level Event (ELE) threat to blockchain systems. Current asymmetric algorithms like RSA and Elliptic Curve Cryptography (ECC), used in Bitcoin, which underpin blockchain security, are particularly vulnerable. A sufficiently powerful quantum computer could break these cryptographic algorithms, potentially leading to catastrophic consequences for the blockchain industry (including cryptocoins), including both permissioned and permission-less networks.

3.2.1. Blockchain's Dependence on ECC

Most, if not all, blockchains rely on ECC (ECDSA to be more exact) to secure transaction validations using digital signatures. This implies that a CRQC can derive the private key of a blockchain transaction from its digital signature and/or the public key: this is equivalent to becoming the owner of the particular cryptocoin in question. Cold wallets are ineffective for this attack vector. With the private key, an attacker can spend or transfer the cryptocoin at will and there will be nothing the real cryptocoin owner can do about it. Whether the blockchain memorializes authenticated cryptocoin transactions, NFTs, or any other digital asset, the vulnerability is the same if the primary security protecting the asset is a digital signature that is not PQC.

3.2.2. Challenges in Cryptographic Transitions

3.2.2.1. Permissioned Blockchains

Permissioned blockchains—those with centralized governance—are relatively better equipped to handle cryptographic transitions. Examples include:

- Central Bank Digital Currencies (CBDCs): Being developed by all G20 nations to modernize payment systems.
- Digital Asset Platforms: Managed by entities like DTCC for secure handling of financial instruments.
- Private Ecosystems: Such as Hedera Hashgraph and JPMorgan, which are used for enterprise-level applications.

In these cases, each entity acting as a central authority can enforce cryptographic updates on a defined timeline, albeit with significant costs and operational disruptions. Integrating the SDFT API into such systems can simplify the initial transition to PQC while ensuring future-proof adaptability. SDFT provides the flexibility to accommodate new cryptographic standards without requiring extensive overhauls repeatedly and in an incremental fashion.

3.2.2.2. Permission-less Blockchains

Permission-less blockchains, such as Bitcoin and Ethereum, operate without a central authority in a consensus protocol, making coordinated cryptographic transitions far more complex. This lack of centralized control, which is a core feature of these networks, complicates efforts to adopt quantum-safe algorithms in an orderly and uniform manner.

When Bitcoin and Ethereum were initially developed, ECC and RSA were the only practical options for digital signatures. As a result, their respective codebases are rigid and lack inherent flexibility to support alternative cryptographic methods. For these systems, cryptoagility—the ability to adapt cryptography dynamically—is not just beneficial but essential.

As we are still living through a quarter century of transitioning from IPv4 to IPv6, for similar reasons, blockchains will suffer the same painful process of realizing the cost of making a design choice based on the best estimates of security at the time. The main difference between the two scenarios is that IPv6 transitions do not secure anything and do not have a ticking countdown clock with an undefined wake-up time.

3.2.3. SDFT's Cryptoagility: A Flexible Solution

SDFT offers a robust solution to the challenges posed by quantum threats by enabling blockchain systems to dynamically integrate and transition between cryptographic algorithms. Its core capabilities include:

1. **Dynamic Algorithm Selection**: SDFT allows each transaction to specify its cryptographic algorithm, embedding these details within the transaction itself.
2. **Self-Describing Transactions**: Transactions become self-describing and self-prescribing, with all necessary cryptographic parameters folded into the SDFT message.
3. **Universal Compatibility**: Node processors equipped with SDFT can process transactions regardless of the chosen cryptographic algorithm, ensuring seamless interoperability.

For permission-less blockchains, this flexibility empowers users to select the level of cryptographic protection they desire for their transactions on their schedule. Coin owners can dynamically choose PQC algorithms, ensuring their assets remain secure without disrupting the broader network.

3.2.4. Representation of Data-At-Rest and Data-In-Transit

SDFT also provides a consistent framework for representing blockchain transactions across their lifecycle:

- Data-At-Rest (DAR): When a transaction is created and stored in a wallet, it is considered DAR.
- Data-In-Transit (DIT): When the transaction is transmitted to a node for validation, it transitions to DIT.
- Data-At-Rest (DAR): Once the transaction is added to the blockchain ledger, it returns to DAR.

By embedding cryptographic parameters directly into the transaction using SDFT, this transaction lifecycle can be managed securely and consistently, without requiring external transitions. SDFT ensures that transactions remain quantum-resistant throughout their journey, safeguarding both individual assets and network integrity.

3.2.5. Future-Proofing Blockchain Systems

The blockchain industry faces significant risks from cryptographic transitions, particularly as quantum threats loom. The CNSA 2.0 timeline published by the NSA targets the Federal Government to transition to PQC by 2033 or earlier [2] which is only 8 years away. SDFT mitigates these risks by enabling:

- Permissioned Blockchains to transition securely to PQC standards with minimal disruption.
- Permission-less Blockchains to adopt cryptoagility, allowing users to dynamically select and implement cryptographic protections when so desired.

SDFT offers an innovative approach to securing blockchain transactions, positioning itself as a critical solution to the evolving landscape of cryptographic threats. By unifying cryptographic processes and providing flexibility, SDFT ensures the long-term viability of blockchain systems in a post-quantum world.

One of its core strengths is its ability to manage Data-At-Rest (DAR) and Data-In-Transit (DIT) in a consistent and unified manner. For example, a transaction begins as DAR when created and stored in a wallet. It transitions to DIT when transmitted to a node for validation and returns to DAR once added to the blockchain ledger. This seamless unaltered representation across the transaction life-cycle fortifies security at every stage while removing unnecessary vulnerability surfaces.

Moreover, SDFT empowers digital asset owners by allowing them to customize the level of cryptographic protection for their digital assets. This combination of robust security against quantum threats and implementation flexibility makes SDFT a transformative framework for safeguarding blockchain systems.

By addressing the challenges of post-quantum cryptography and offering a forward-compatible solution, SDFT not only secures current systems but also prepares them for the future, ensuring the enduring integrity of blockchain networks for the foreseeable future.

3.3.Data-Centric Security

3.3.1. NUTS needed SDFT

SDFT was not conjured out of thin air. It was needed in the research and development of the NUTS Ecosystem and specifically the nut capsule, a secure encapsulation for data with unique capabilities. Originally, two or three iterations of the NUT API were written in Python. But it became quickly obvious that choices had to be made in selecting various cryptographic algorithms and their associated parameters for the API. The questions became very clear: what if we made a weak choice? What if a different user wanted something else? What about the old nut capsule on disk with old choices? Does that mean we have to maintain different versions of the read/write routines? Then, we encountered the classic problem of the code being intertwined with the choice of cryptography and parameters.

One of the reasons why applied cryptography is hard to abstract out to higher levels of design is that the details drag you down to making difficult decisions which inevitably is expressed in ways that limit the flexibility of the codebase and encrypted data formats. Thus, SDFT was created out of the need to allow abstractions using cryptographic methods and make the codebase and data independent of the cryptography choices as much as possible. This focus on allowing abstractions resulted in what SDFT looks like today.

3.3.2. NUTS: What is Data-to-Data communications?

Thus far, the development of the Internet has been about machine-to-machine communications.

The Next Generation Internet will be about data-to-data communications.

NUTS focuses on these goals: how to manage secured data throughout its lifecycle automatically with minimal administrative overhead; and how to allow data to communicate with each other.

IP addresses, MAC addresses, ports, process IDs: these are all parts of the current Internet to identify a particular process running a specific application allowing connections from a different application running on a different process on a different port. But it stops there, application-to-application or machine-to-machine, there's not that much difference. Data-centric approaches have various meanings from different studies (ACDC [35], TDF/ZTDF [36]) and different vendors such as Virtru [37] and Seclore [38]. Data-centric security approaches also present a cacophony of definitions and sometimes called Zero Trust Data.

NUTS, or eNcrypted Userdata Transit & Storage or NUTS [11], envisions and enacts a digital environment where data can,

- Protect itself
- Identify itself
- Recognize its owner
- Converse with itself and other data
- Ask for authenticated services to be performed on itself
- Replicate itself
- Heal itself
- Know its provenance and history
- Preserve itself

- Realize when tampered with
- Manage itself
- Plan a future for itself

The characteristics listed above are derived from a study of viewing DNA as a data model and applying its operating principles upon digital data. NUTS implements such an ecosystem where security is NOT an afterthought but integrated at the very THING that all systems want to protect, access to data.

In NUTS, a secure data encapsulation called a nut capsule is created from a complex data structure where many sub parts are created from SDFT messages. A nut capsule is endowed with an independently created identifier called a NutID at creation. The security layers of the nut capsule present novel portable cryptographic technical controls to give fine grain access controls back to the data owner.

The NUTS Ecosystem enables unlimited data sizes, location independence, security portability (think BYOS: "bring your own security"), and data portability. And, of course, a nut capsule inherits all the capabilities of SDFT messages such as easy transitions of cryptography and unified DAR and DIT. We found that building entire ecosystems out of nut capsules allows the security of the nut capsule to percolate to higher layers of the system, we call this ***fractal security***.

Identifiers have been around for a long time. What's interesting is that most identifiers are doled out from a central authority primarily for uniqueness, control, and de-anonymization. The NUTS data-centric model flips that on its head to allow independent user devices to come up with NutIDs with large enough entropy that collisions are minimized. No other system we know of allows that level of specificity to a piece of data that a single user might create using their own laptop. But this type of anonymized and random identifier is produced by DNA every day, there is no registry in Nature, and theoretically, every person who has ever been born could be identified by their DNA fingerprint. We opted to not philosophize on why Nature creates unique identifiers, but to assume that an ID is a significant factor in its data model.

The current rage in the cybersecurity industry is about Zero Trust Identity Access Management. This is a narrow-focused implementation of data-centricity in only trying to map authorized accesses to digital assets by identified and authenticated users. A user is represented as identifiable data in any Zero Trust IAM system. Extrapolated further, a piece of data can be represented as identifiable data in a data-centric system. The additional condition of data-centric *security* requires the identifiable data to be represented in a secure encapsulation such as a nut capsule.

The ability to identify a user's one piece of data on any connected storage anywhere in the world is a unique proposition when coupled with security that only allows the user to access the data and send messages to the data in order to manipulate it. This 'data' can be anything, scripts, documents, databases, applications, images, credentials, LLMs, configurations, IoT devices, printers, systems, memory, networks, satellites, data streams, sensors, etc. NUTS allows data to be acted upon directly in an authenticated manner no matter where it is stored. We have found this paradigm shift of directly communicating with data having profound implications on how systems can be designed to be inherently secure and ultimately provide more utility for the user.

For example, the NutManager, a user facing interface to the NUTS Ecosystem, implements a simple nut-based chat session where a single chatnut is created and shared (via a Data Defined Network) amongst the participants and the payload is a single line of text. Since a nut capsule is designed to carry a configurable amount of revision history of the payload, the revision history turns out to be the actual chat session. Adding to the revision history just requires a group member to modify the payload and the NUTS

Ecosystem 'sends a message' to the chatnut to synchronize the change in each copy of the chatnut sitting on each members' laptops. In the prototype nutchat implementation, chatnuts are transmitted in whole since they are small in size. Each chatnut is eventually made consistent by each group member when received. This is an implementation of generalized eventual consistency at the object level with choice of collision algorithm per payload type with full security. We do not know of any other system with this combination of capabilities.

Essentially, the nutchat provides end-to-end encrypted chat sessions with full encrypted chat histories on each members' devices with unlimited participants, all without a dedicated chat server sitting in the cloud: the NUTS Ecosystem just moves nut capsules. This example shows that every nut capsule has built-in eventual consistency with configurable histories, an inspiration from the DNA data model.

Imagine an operating system that will only execute an application that comes wrapped in a nut capsule? This simple concept can eliminate most malware from running on systems. In a secured data-to-data communication channel, it is possible to deterministically eliminate all spam and intruders; think about a secure nut-based email system where it's virtually impossible to receive spam. The current production version of the NUTS Ecosystem is capable of providing near real-time recovery of nut capsules for groups of trusted users (Data Defined Networks or FHOGs [39]) with full revision control capabilities at the object level on each users' local device. This type of feature set in a single platform is unheard of without a large IT administrative team supporting it. The NUTS Ecosystem runs itself.

How to manage the keys? The NUTS Ecosystem provides a user interface called the NutManager with a built-in universal KMS. Any KMS requires 3 features: secure storage for the key, key identification, and key distribution. By putting a key inside a nut capsule which is acted upon by the NUTS Ecosystem, all three requirements of the KMS are satisfied. NUTS is the only data management system where the KMS is identical to the data protection system that the keys are used for. Not only does this design provide utter simplicity, but it is also scalable, and more importantly, it removes the need for a dedicated IT administrator to manage the KMS and/or data protection system which thus removes another possible attack vector and cost. We do not believe a business modeled on keeping a client's keys hostage on a centralized server is fair, proper, economical, nor secure. Therefore, NUTS provides a fully independent KMS on every user installation.

SDFT plays an integral role in constructing secure encapsulations like a nut capsule. NUTS is secure by design, secure by default [40] [41], and provides easy transitions to new cryptography due to its inherited features of SDFT for both DAR and DIT. With all the cybersecurity problems on the Internet, NUTS was designed to address the root causes of those problems rather than to temporarily relieve symptoms. When security becomes integrated with the data, some symptoms never even appear. NUTS can enable data-to-data communications to usher in a new age of digital networks where the primary focus is on protecting your data all the time, everywhere.

4. Acknowledgements

SDFT and NUTS were made possible with the patience and support of colleagues, family and friends. This research of integrating PQC algorithms into SDFT was performed under the following financial assistance award 70NANB24H069 from U.S. Department of Commerce, National Institute of Standards and Technology. We thank Noah Waller and Andrew Regenscheid for their helpful guidance and perspectives during the project period. We would also like to thank Jose Colucci, Kiara Farmer, Nicole Berry, Jacqueline Gray, and Ashley Smothers for their helpful guidance through the world of NIST/Government grant administrative processes.

5. References

[1] National Institute of Standards and Technology, "NIST Releases First 3 Finalized Post-Quantum Encryption Standards," 13 Aug 2024. [Online]. Available: https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards.

[2] National Security Agency, "CNSA 2.0 Announcing the Commercial National Security Algorithm Suite 2.0," Sep 2022. [Online]. Available: https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF.

[3] The White House, "NSM-10 National Security Memorandum on Promoting United States Leadership in Quantum Computing While Mitigating Risks to Vulnerable Cryptographic Systems," 4 May 2022. [Online]. Available: https://www.whitehouse.gov/briefing-room/statements-releases/2022/05/04/national-security-memorandum-on-promoting-united-states-leadership-in-quantum-computing-while-mitigating-risks-to-vulnerable-cryptographic-systems/.

[4] 117th U.S. Congress, "H.R.7535 - Quantum Computing Cybersecurity Preparedness Act," 21 Dec 2022. [Online]. Available: https://www.congress.gov/bill/117th-congress/house-bill/7535.

[5] National Cybersecurity Center of Excellence (NCCoE), "MIGRATION TO POST-QUANTUM Cryptography (PQC)," Aug 2023. [Online]. Available: https://www.nccoe.nist.gov/sites/default/files/2023-08/mpqc-fact-sheet.pdf.

[6] A. K. Lenstra, "The Three Pillars of Cryptography," 2008.

[7] W. Newhouse, M. Souppaya, W. Barker and C. Brown, "NIST SP1800-38A,, Migration to Post-Quantum Cryptography: Preparation for Considering the Implementation and Adoption of Quantum Safe Cryptography, Volume A," National Institute of Standards and Technology, Rockville, MD, 2023.

[8] W. Newhouse and al, "NIST SP 1800-38B, Migration to Post-Quantum Cryptography Quantum Readiness: Cryptographic Discovery, Volume B," National Institute of Standards and Technology, Rockville, MD, 2023.

[9] W. Newhouse and al, "NIST SP 1800-38C, Migration to Post-Quantum Cryptography Quantum Readiness: Testing Draft Standards, Volume C," National Institute of Standards and Technology, Rockville, MD, 2023.

[10] G. Alagic and al, "NIST IR 8528, Status Report on the First Round of the Additional Digital Signature Schemes for the NIST Post-Quantum Cryptography Standardization Process," National Institute of Standards and Technology, Gaithersburg, MD, 2024.

[11] Y. H. Auh, "NUTS: eNcrypted Userdata Transit and Storage". US Patent 10671764, 2 June 2020.

[12] T. Brown, "Design Thinking," Harvard Business Review, Cambridge, MA, 2008.

[13] Y. H. Auh, "Structured Data Folding with Transmutations". US Patent 10503933, 10 Dec 2019.

[14] M. Jones, "JSON Web Key (JWK)," Internet Engineering Task Force (IETF), 2015.

[15] H. Okhravi,, T. Hobson, D. Bigelow and W. Streilein, "Finding Focus in the Blur of Moving-Target Techniques," IEEE, 2014.

[16] C. Lei and al, "Moving Target Defense Techniques: A Survey," Wiley, NY, 2018.

[17] R. Banks, "Cybersecurity: 5th Generation of Security," in *MIT*, Cambridge, MA, 2022.

[18] N. Alnahawi, N. Schmitt, A. Wiesmaier, A. Heinemann and T. Grasmeyer, "On the State of Crypto-Agility," Cryptology Archive, 2022.

[19] Information Technology Laboratory, "FIPS 197 ADVANCED ENCRYPTION STANDARD (AES)," National Institute of Standards and Technology, Gaithersburg, MD, 2021.

[20] Information Technology Laboratory, "Block Cipher Techniques," National Institute of Standards and Technology, 2017. [Online]. Available: https://csrc.nist.gov/Projects/block-cipher-techniques/BCM.

[21] Information Technology Laboratory, "Block Cipher Techniques," National Institute of Standards and Technology, 2017. [Online]. Available: https://csrc.nist.gov/projects/block-cipher-techniques.

[22] Information Technology Laboratory, "Block Cipher Techniques: Modes Development," National Institute of Standards and Technology, 2017. [Online]. Available: https://csrc.nist.gov/projects/block-cipher-techniques/bcm/modes-development.

[23] K. S. Perumalla, "Introduction to Reversible Computing," CRC Press, 2013.

[24] R. Landauer, "Irreversibility and Heat Generation in the Computing Process," IBM Journal of Research and Development, Yorktown Heights, NY, 1961.

[25] C. H. Bennett, "Logical Reversibility of Computation," IBM Journal of Research and Development, Yorktown Heights, NY, 1973.

[26] Information Technology Laboratory, "FIPS 204 Module-Lattice-Based Digital Signature Standard," National Institute of Standards and Technology, Gaithersburg, MD, 2024.

[27] Information Technology Laboratory, "FIPS 205 Stateless Hash-Based Digital Signature Standard," National Institute of Standards and Technology, Gaithersburg, MD, 2024.

[28] Open Quantum Safe, "open-quantum-safe/liboqs-python," Open Quantum Safe, 2024. [Online]. Available: https://github.com/open-quantum-safe/liboqs-python.

[29] Open Quantum Safe, "liboqs," Open Quantum Safe a Series of LF Projects, LLC, 2024. [Online]. Available: https://openquantumsafe.org/liboqs/.

[30] Information Technology Laboratory, "Post-Quantum Cryptography PQC," National Institute of Standards and Technology, 2024. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography.

[31] Information Technology Laboratory, "FIPS 203 Module-Lattice-Based Key-Encapsulation Mechanism Standard," National Institute of Standards and Technology, Gaithersburg, MD, 2024.

[32] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[33] Open source project, "src/secp256k1," 2024. [Online]. Available: https://github.com/bitcoin/bitcoin/tree/master/src/secp256k1.

[34] Bitcoin community, "Signatures," 2024. [Online]. Available: https://en.bitcoin.it/wiki/Protocol_documentation#Signatures.

[35] Lincoln Laboratory MIT, "Automated Cryptography for Data-Centric Security (ACDC)," 2019. [Online]. Available: https://www.ll.mit.edu/r-d/projects/automated-cryptography-data-centric-security.

[36] Virtru, "Zero Trust Data Format: The Quiet Revolution in Secure Collaboration for Defense & Intelligence," 2024. [Online]. Available: https://www.virtru.com/blog/zero-trust/zero-trust-data-format-the-quiet-revolution-in-secure-collaboration-for-defense-intelligence.

[37] Virtru, "TDF: The Standard Transforming Data-Centric Security," 2024. [Online]. Available: https://www.virtru.com/data-security-platform/trusted-data-format.

[38] Seclore, "Enhancing RBI Compliance with Seclore's Data-Centric Security," 2024. [Online]. Available: https://www.seclore.com/wp-content/uploads/Infographic-Enhancing-RBI-Compliance-with-Seclores-Data-Centric-Security.pdf.

[39] Y. H. Auh, "NUTS: Flexible Hierarchy Object Graphs". US Patent 11558192, 17 Jan 2023.

[40] A. Cavoukian, "Privacy by Design in law, policy and practice : Privacy by Design," Canadian Electronic Library, Canada, 2011.

[41] CISA, "Secure-by-Design: Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Secure by Design Software," CISA, 2023.